

DESAN: READ/WRITE-Aware Redundant Sanitizer Check Elimination

Yunlong Xing
Michigan Technological University
Houghton, Michigan, USA
George Mason University
Fairfax, Virginia, USA
yuxing@mtu.edu

Shiyu Sun
George Mason University
Fairfax, Virginia, USA
ssun20@gmu.edu

Kun Sun
George Mason University
Fairfax, Virginia, USA
ksun3@gmu.edu

Abstract

Sanitizers such as ASan, UBSan, and MSan are widely used to detect memory-related errors and undefined behavior. However, their high runtime overhead makes continuous use difficult in testing and production-like executions. A major source of this overhead is redundant sanitizer instrumentation: compiler-inserted checks often validate the same variable repeatedly along the same control-flow paths. Reducing sanitizer checks requires reasoning about the property each check validates and the program state on which it depends. We explore a conservative policy that eliminates only covered READ checks and retains WRITE checks and those whose semantics remain unresolved. This paper presents DESAN, an LLVM IR-level framework that reduces redundant sanitizer checks while preserving sanitizer semantics. DESAN identifies high-impact core checks and recovers the checked variable for each check through backward IR tracing. It then classifies checks as READ, WRITE, or UNKNOWN and builds per-variable check graphs to find covered READ checks. DESAN preserves all WRITE and UNKNOWN checks. When safe, it removes redundant READ checks along with their sanitizer-only IR slices, including shadow computation, comparisons, branches, and isolated report paths. We evaluate DESAN on SPEC CPU2006, SPEC CPU2017, and 20 open-source projects. Across 163 valid sanitizer configurations, core checks cover a median of 96.62% of inserted checks, and DESAN removes a median of 51.48% of all checks while preserving evaluated bug-detection capability. Compared with ASAN— for ASan and SanRazor for UBSan, DESAN achieves competitive or stronger overhead reduction.

CCS Concepts: • Security and privacy → Software security engineering.

Keywords: READ/WRITE Semantics, Sanitizer Checks, Redundancy Elimination.

ACM Reference Format:

Yunlong Xing, Shiyu Sun, and Kun Sun. 2026. DESAN: READ/WRITE-Aware Redundant Sanitizer Check Elimination. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026)*, November 15–18, 2026, Hong Kong, China. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3828523.3857029>

1 Introduction

Sanitizers detect runtime errors such as buffer overflows, use-after-free errors, uninitialized memory uses, type mismatches, and other undefined behaviors [15, 16, 24, 25, 27, 28]. They typically work by inserting additional instrumentation during compilation and validating program operations at runtime. When a violation is detected, the sanitizer reports the error type and program location, helping developers reproduce and diagnose security-critical bugs. AddressSanitizer (ASan) [14], UndefinedBehaviorSanitizer (UBSan) [16], and MemorySanitizer (MSan) [15] are widely used in testing, fuzzing, and debugging because they find many bugs with relatively little deployment effort.

Despite their effectiveness, sanitizers impose substantial performance overhead, which limits their use in large-scale testing and production-like executions. The overhead comes from more than a runtime reporting call. A sanitizer check may compute an address or shadow state, load metadata, branch on the result, and enter a cold report path if the check fails. Because compiler instrumentation inserts these checks mechanically, they often validate the same property repeatedly without considering program semantics. Therefore, sanitizer-instrumented programs can execute many redundant checks before reaching the original program operation. Prior reports and official documentation show that ASan commonly incurs around a 2× slowdown [14, 23], UBSan adds noticeable overhead on many programs [16, 35], and MSan can slow execution by several times because it tracks shadow state for uninitialized values [15]. Sanitizer overhead matters in systems testing because execution time limits the number of test inputs and configurations that can be exercised within a fixed resource budget. Reducing this cost can make sanitizer-enabled testing more practical in CI, fuzzing, and production-like validation.

Prior systems reduce sanitizer overhead through profile-guided selection and static optimization. ASAP and SanRazor



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11

<https://doi.org/10.1145/3828523.3857029>

use execution profiles to reduce checks [29, 35]. ASan— statically debloats ASan instrumentation [36]. These approaches demonstrate that substantial instrumentation costs can be avoided. DESAN takes a complementary approach. It eliminates only READ checks and uses checked-property coverage to identify redundant ones. Its contribution lies in connecting this policy to the low-level instrumentation that implements each check, rather than in claiming that WRITE checks are never redundant.

READ and WRITE identify the program operations associated with checks, but they do not determine whether two checks are equivalent. For example, ASan load and store checks can validate the same addressability property over the same memory range. Either type may be redundant under suitable conditions. A repeated READ check may still be necessary if the accessed range, object lifetime, or relevant sanitizer state changes.

We use a restricted transformation policy. It retains WRITE and UNKNOWN checks and considers only READ checks for elimination. Retaining WRITE checks preserves the validation for each instrumented write. Retaining UNKNOWN checks avoids transformations when access semantics cannot be recovered. This policy limits the scope of optimization. A READ check may be removed only when it validates the same property, every incoming path is covered, and no relevant state has been invalidated.

In this paper, we present DESAN, an LLVM IR-level framework for reducing redundant sanitizer checks while preserving sanitizer semantics. DESAN runs after sanitizer instrumentation and before final binary generation. It optimizes only semantically recoverable core checks. For each such check, DESAN recovers the checked variable, classifies the access as READ, WRITE, or UNKNOWN, and removes only READ checks covered by previous checks in the per-variable control-flow graph. The optimized binary continues to use the original sanitizer runtime.

DESAN operates in three stages. First, it identifies *core checks*: frequent sanitizer check families that directly guard program accesses or value uses and whose checked operands can be recovered from LLVM IR. These include ASan load/store checks, major UBSan handlers such as type-mismatch and pointer-overflow checks, and common MSan checks for warnings, parameters, return values, and shadow state. Second, DESAN traces check operands backward through common IR patterns, including load, store, getelementptr, casts, phi, and select. It then normalizes equivalent checked variables and records their access type. Third, DESAN constructs a per-variable check graph and removes covered READ checks. When DESAN removes a redundant check, it also deletes the associated sanitizer-only IR slice when that code has no non-sanitizer uses. This includes the report call, shadow computation, comparisons, branches, and isolated report blocks. The cleaned IR is then simplified and recompiled into a sanitizer-enabled binary.

We evaluate DESAN on SPEC CPU2006, SPEC CPU2017, and 20 widely used open-source projects. After filtering unsuccessful configurations and configurations with no checks, the evaluation contains 163 valid sanitizer configurations across ASan, UBSan, and MSan. The selected core checks cover most inserted sanitizer checks, with a median Core/All ratio of 96.62%. DESAN removes a median of 51.48% of all inserted sanitizer checks while preserving all WRITE and UNKNOWN checks. For example, on SPEC CPU2017, it removes 55.08% of ASan checks, 41.07% of UBSan checks, and 78.38% of MSan checks. The corresponding sanitizer overhead reductions are 50.81%, 48.00%, and 34.04%. On 20 open-source projects, DESAN removes 51.02% of ASan checks, 42.91% of UBSan checks, and 72.52% of MSan checks. Compared with ASAN— for ASan and SanRazor for UBSan, DESAN achieves competitive or stronger overhead reduction under a more conservative policy that removes only READ checks. We also evaluate bug-detection preservation using 7,460 bugs from Juliet Test Suite [20] and 108 reproducible real-world vulnerabilities [5, 18]. All bugs detected by the original sanitizers remain detectable after applying DESAN. The source code, scripts, and experimental artifacts of DESAN are publicly available at <https://github.com/YunlongXing/DESAN>.

In summary, this paper makes the following contributions:

- We distinguish the READ/WRITE preservation policy from the conditions that establish check redundancy, and formulate READ-check elimination in terms of checked properties, coverage, and invalidation.
- We present a core-check analysis for frequent sanitizer checks across ASan, UBSan, and MSan whose checked variables can be recovered from LLVM IR.
- We design an LLVM IR-level framework that recovers checked variables, builds per-variable check graphs, removes covered READ checks, and deletes the associated sanitizer-only IR slice.
- We implement a prototype of DESAN and evaluate it on SPEC CPU2006, SPEC CPU2017, 20 open-source projects, Juliet bugs, and real-world CVEs. The results show substantial sanitizer-check reduction while preserving the evaluated bug-detection capability.

2 Preliminaries

Sanitizers are compiler-based runtime tools that detect unsafe program behavior, such as invalid accesses, misaligned values, and uses of uninitialized data. They instrument checks into a program at compile time and validate the corresponding properties at runtime [4, 14–16, 28]. This paper focuses on three widely used LLVM sanitizers: AddressSanitizer (ASan), UndefinedBehaviorSanitizer (UBSan), and MemorySanitizer (MSan) [9, 30]. ASan uses shadow memory and redzones to detect memory errors such as out-of-bounds accesses and use-after-free. UBSan detects undefined behavior, including misaligned accesses, invalid type uses, pointer

overflow, and out-of-bounds indexing. MSan detects uses of uninitialized values by propagating shadow state.

Sanitizer overhead. Sanitizers are effective but expensive. ASan typically incurs about a 2x slowdown [14], UBSan introduces substantial overhead in practice [35], and MSan can slow programs down by 3–5x [15]. The overhead comes from two sources. First, each inserted check expands a source-level access into several IR operations: computing the address, mapping it to shadow state, accessing shadow memory, testing the result, branching, and possibly calling the runtime. Second, sanitizers add checks locally and conservatively, so nearby CFG paths or repeated accesses to the same variable may check the same property more than once.

Sanitizer check as an IR slice. In this paper, a sanitizer check is not merely the call to a sanitizer runtime function. The check is the full IR slice inserted to validate a property and report a violation. Figure 1 illustrates an ASan load check for `bea->tail`. Lines 1–3 compute the program address being checked. Lines 4–6 map that address to ASan shadow memory, and Lines 7–9 load and test the shadow byte before branching to the error path. Lines 10–12 form the report path. The call to `__asan_report_load8` is only the cold-path reporting call; most hot-path overhead comes from shadow mapping, the shadow load, the comparison, and the branch. Therefore, an optimization that disables only the report call may leave most sanitizer overhead in the program.

```

1  %0 = load ptr, ptr %bea.addr, align 8
2  %tail = getelementptr %struct.arc, ptr %0, i32 0, i32 2
3  %1 = ptrtoint ptr %tail to i64
4  %2 = lshr i64 %1, 3
5  %3 = add i64 %2, 2147450880
6  %4 = inttoptr i64 %3 to ptr
7  %5 = load i8, ptr %4, align 1
8  %6 = icmp ne i8 %5, 0
9  br i1 %6, label %7, label %8
10 %7:
11 call void @__asan_report_load8(i64 %1) #5
12 unreachable

```

Figure 1. An ASan check is an IR slice, not only a runtime call.

Read and write checks. READ and WRITE describe the access associated with a sanitizer check. `__asan_report_load*` reports invalid reads, and `__asan_report_store*` reports invalid writes, in ASan. The access kind determines the reported operation, but it does not establish a general ordering between the properties checked. In particular, ASan’s load/store distinction should not be interpreted as a separate test of operating-system read/write permissions.

If a READ and a WRITE check validate the same address range while the relevant state remains unchanged, the WRITE check may also be redundant in principle. Nevertheless, DeSan retains WRITE checks as a conservative policy. It eliminates only READ checks whose coverage can be established. Retaining WRITE checks also does not make subsequent READ checks automatically removable. The guarded

write or another operation may invalidate the property on which that coverage depends. For UBSan and MSan, the checked predicate or initialization state must likewise be assessed independently of the READ/WRITE classification.

Implication for sanitizer check reduction. DeSan distinguishes eligibility from redundancy. Under its policy, only READ checks are eligible for elimination. A candidate must also satisfy the property-specific coverage condition below. WRITE and UNKNOWN checks are retained even if a more general analysis could establish that they are redundant.

Redundancy condition. A repeated variable reference does not imply a redundant check. A READ check may be eliminated only when every execution path that reaches it has already validated the same sanitizer property for the same object or value. The property-relevant parameters match, and no intervening event may invalidate the earlier validation. Different paths may provide different covering checks. READ/WRITE classification determines eligibility under our removal policy. It does not establish property equivalence.

3 A Motivating Example

We use a simplified tree-update routine to show how sanitizer checks can become redundant after instrumentation and how DeSan removes them without treating all checks alike. The example uses ASan, but the same high-level workflow also guides our treatment of UBSan and MSan in later sections.

How ASan inserts checks. Figure 2(a) shows the original program fragment. It repeatedly accesses fields such as `bea->tail`, `temp->sibling`, and `temp->sibling_prev`. When compiled with ASan, each pointer dereference is guarded by a check. For readability, Figure 2(b) represents each full ASan check slice with a compact marker: `[read]` for a read check and `[write]` for a write check. Each marker stands for the full sanitizer instrumentation slice described in Section 2: the shadow-memory access, the check, the control-flow branch, and the report path.

ASan distinguishes load and store checks according to the protected program action. For example, reading `temp->pred` requires a READ check, while assigning to `temp->pred` requires a WRITE check immediately before the store. Assignments often require both kinds of checks: the right-hand side is a read, and the left-hand side is a write. Nested field accesses can require multiple checks because ASan first validates the base object before checking the subfield. This small fragment contains 21 ASan checks: 15 READ checks and 6 WRITE checks.

Where redundancy appears. The instrumentation is local and conservative, so it repeatedly checks the same variable along nearby control-flow paths. For instance, `bea->tail` is checked twice in the same condition. The second READ does not validate a new program object: it is covered by the first check of the same field on the same path. Several accesses to `temp->sibling` and `temp->sibling_prev` likewise

<pre> 1 void update_tree(...) { 2 if(bea->tail == jplus bea->tail == iplus) 3 sigma = ABS(sigma); 4 father = temp->pred; 5 while(temp != jminus) { 6 if(temp->sibling != temp->sibling_prev) 7 temp->sibling_prev->sibling = temp->sibling; 8 else 9 father->child = temp->sibling; 10 temp->pred = new_pred; 11 temp->sibling = new_pred->child; 12 if(temp->sibling) 13 temp->sibling->sibling_prev = temp; 14 temp->sibling_prev = 0; 15 if(orientation_temp == cycle_ori) 16 flow_temp = temp->flow + delta; 17 else 18 flow_temp = temp->flow - delta; 19 basic_arc_temp = temp->flow; 20 father = temp->pred; 21 } 22 } </pre>	<pre> 1 void update_tree(...) { 2 if(bea->tail [read] bea->tail [read]) 3 ... 4 temp->pred [read] 5 while(...) { 6 if(temp->sibling [read], temp->sibling_prev [read]) 7 temp->sibling[read], temp->sibling_prev[read], temp->sibling_prev->sibling[write] 8 else 9 temp->sibling [read], father->child [write] 10 temp->pred [write] 11 new_pred->child [read], temp->sibling [write] 12 if(temp->sibling [read]) 13 temp->sibling [read], temp->sibling->sibling_prev [write] 14 temp->sibling_prev [write] 15 if(...) 16 temp->flow [read] 17 else 18 temp->flow [read] 19 temp->flow [read] 20 temp->pred [read] 21 } 22 } </pre>
(a) Original code	(b) ASan check review and redundant READ removal

Figure 2. A motivating example of repeated sanitizer checks. Each [read] and [write] marker represents an ASan check slice associated with the corresponding access. Struck-through [read] checks are covered by earlier checks and can be removed by DESAN. WRITE checks are retained.

occur after earlier checks for the same variables. At the merge after the if-else over `temp->flow`, both incoming paths have already checked `temp->flow`, so the later READ at the merge is also covered. The relevant distinction is between changing a stored value and changing the property checked by the sanitizer. For example, assigning to a pointer field changes the pointer used in a later dereference, even though the field’s own address may remain unchanged. Consequently, coverage of a read from the field does not establish coverage of access to the object it newly references.

All WRITE checks in Figure 2(b) are retained under DESAN’s preservation policy. READ-removal requires a separate argument. The earlier validation must concern the same property and access range, and it must remain valid along every path to the later check. A preceding WRITE check can provide this validation only if the following store and other intervening operations do not invalidate it.

How DESAN reduces the example. DESAN first recovers the checked variable for each sanitizer check by tracing the checked operand backward from the report call to the original address computation. For ASan, this often means tracing from the report argument through `ptrtoint` and `getelementptr` back to a normalized access path. DESAN then groups checks by the normalized checked variable and builds a per-variable check CFG that records the READ/WRITE type and the relevant control-flow coverage.

The graph organizes candidate READ checks for coverage analysis. The second check of `bea->tail` illustrates a linear candidate, while the later check of `temp->flow` illustrates a candidate at a control-flow merge. Elimination additionally requires matching checked properties and access parameters, with no relevant invalidation. In particular, repeated

field names alone do not justify removal across stores or potentially aliasing updates. Once redundancy is established, the associated sanitizer-only slice can be removed and the resulting dead code simplified.

4 Core Checks

Sanitizers use different checks to detect and report errors, but not all of them add significant performance overhead. Some also act as safer wrappers for existing functions and should remain unchanged. This matches our goal: remove only redundant checks without changing the original code logic. We examine the static distribution of sanitizer checks across projects and select frequent check families whose guarded operands can be recovered from LLVM IR.

DESAN treats a sanitizer check as a sanitizer-inserted LLVM IR instrumentation unit. The unit is anchored by a runtime call, report handler, or explicit check instruction, and validates a program access, a pointer or value use, or the relevant shadow state. Since DESAN runs after sanitizer instrumentation, these checks are already explicit in the IR and can be inspected uniformly across compilation units. For each module, DESAN scans the call sites and groups sanitizer-related callees by sanitizer family. It then counts how often each check type appears. For a sanitizer S , the static ratio of a check type c is computed as

$$\text{Ratio}(c, S) = \frac{\text{Count}(c)}{\sum_{c' \in S} \text{Count}(c')}.$$

DESAN selects core checks from the most frequent check types whose checked operands can be recovered semantically. A check is considered a core check when it directly guards a program access or value use, and its checked operand

can be traced to a normalized IR variable. Redundant instances can also be removed by deleting sanitizer-only instrumentation without changing the original semantics.

Table 1. Core sanitizer check families analyzed by DESAN.

Sanitizer	Core check type
ASan	<code>__asan_report_load*</code> , <code>__asan_load*</code> <code>__asan_report_store*</code> , <code>__asan_store*</code>
UBSan	<code>__ubsan_handle_type_mismatch_v1</code> <code>__ubsan_handle_pointer_overflow</code> <code>__ubsan_handle_out_of_bounds</code> <code>__ubsan_handle_shift_out_of_bounds</code>
MSan	<code>__msan_warning*</code> , <code>__msan_maybe_warning*</code> <code>__msan_param_*</code> , <code>__msan_retval_*</code> , <code>__msan_va_arg_*</code> <code>__msan_check_mem_is_initialized</code> , <code>__msan_test_shadow</code>

For ASan, the core checks are memory load and store checks, in both report-style and callback-style forms, as shown in Table 1. Report-style checks provide useful anchors for recovering the checked variable. The report argument is the checked address, which DESAN traces backward through common ASan instrumentation patterns, including `ptrtoint`, shadow-address arithmetic, casts, and the original `getelementptr`. DESAN treats ASan load checks as READ checks and ASan store checks as WRITE checks. It does not optimize ASan wrapper routines such as `__asan_memcpy`, `__asan_memmove`, and `__asan_memset`. These routines replace or wrap existing memory operations and carry original program behavior in addition to checking behavior, so rewriting them would go beyond redundant-check removal.

UBSan and MSan checks require more conservative handling because their runtime calls do not always correspond to simple memory reads or writes. UBSan handlers represent type mismatch, pointer arithmetic, bounds, or shift checks. The same handler family can protect different source-level operations depending on its operands and nearby IR uses. MSan checks often validate shadow state for values passed through ordinary arguments, return values, variadic arguments, or explicit memory-initialization queries. Therefore, DESAN infers READ, WRITE, or UNKNOWN from the check kind, backward tracing, and local IR context. Only checks classified as READ participate in redundant read-check elimination. WRITE checks are always preserved, and UNKNOWN checks are kept unchanged unless static analysis can safely refine them to READ.

Figure 3 shows that the selected core checks cover most inserted checks in our SPEC experiments. In SPEC CPU2006, core checks account for 94.05% of ASan checks, 84.84% of UBSan checks, and 100.00% of MSan checks. In SPEC CPU2017, the corresponding ratios are 95.86%, 87.31%, and 100.00%. These results show that the selected families account for

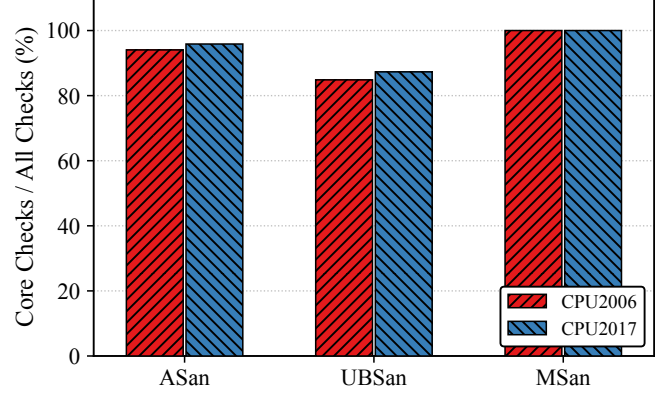


Figure 3. Core checks cover most sanitizer checks in SPEC CPU2006 and SPEC CPU2017.

most statically counted checks in the evaluated configurations. This motivates DESAN’s design: rather than transforming every sanitizer runtime call, DESAN focuses on frequent, semantically recoverable core checks and leaves non-core checks untouched as a conservative safety boundary.

5 System Design

DESAN is an LLVM IR-level sanitizer check reduction pass. It runs after ASan, UBSan, or MSan has instrumented the program and before the compiler generates the binary. This placement is important: sanitizer checks are explicit in the IR, while the original program logic is still visible enough for data-flow and control-flow analysis. DESAN does not replace sanitizer runtimes or change sanitizer semantics globally. Instead, DESAN finds redundant sanitizer read checks and removes the corresponding sanitizer-only IR. The standard compiler pipeline then generates the final binary.

Figure 4 shows the overall workflow. The source program is first compiled with sanitizer instrumentation enabled. DESAN then performs three major stages. It first identifies checked variables: it collects the core checks, recovers the variables they guard, and builds a per-variable check CFG. Next, it removes read checks already covered by previous checks for the same variable. Finally, it deletes the associated sanitizer-only instruction slice, simplifies the IR, and recompiles the optimized IR into a sanitizer-enabled binary.

5.1 Checked Variables

The first stage turns low-level sanitizer calls into variable-level facts. A sanitizer report call alone is not enough to decide redundancy. DESAN must know which object the check protects, whether the access is a read or a write, and how the check relates to other checks in the CFG. This includes three steps: it identifies core checks, recovers the checked variables, and constructs the per-variable check CFG.

Core check identification. DESAN starts by scanning the instructions in each non-declaration function and selecting

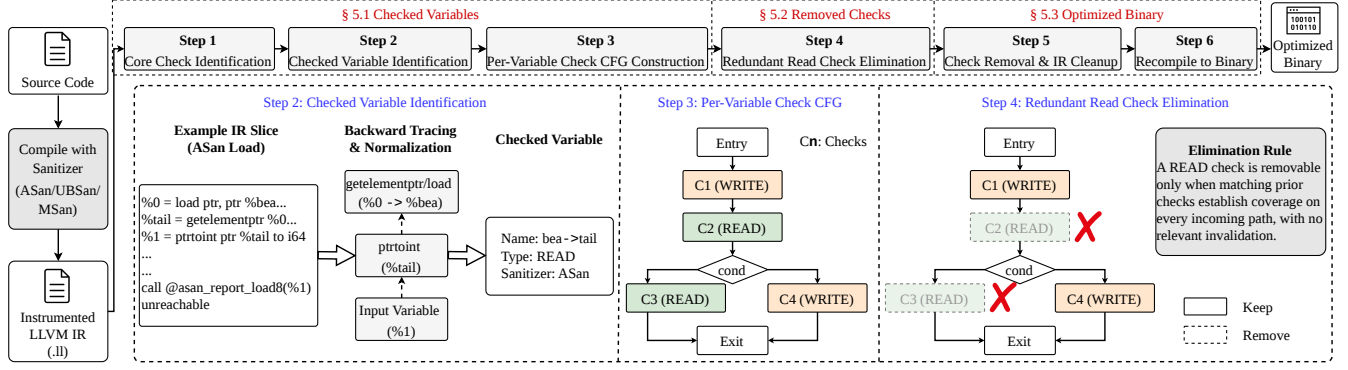


Figure 4. Overview of DESAN's sanitizer check reduction pipeline.

calls that come from sanitizer instrumentation. The collector normalizes callee names and classifies each call as ASan, UBSan, MSan, or an unknown sanitizer family. The collector then keeps only the calls that match the core-check set described in Section 4. This avoids reasoning about runtime helpers whose semantics are not simple checks, such as sanitizer wrappers around memory library functions.

The collector records global statistics and per-function check instances. The global statistics report the distribution of sanitizer checks and confirm that the selected core checks cover the dominant part of the instrumentation. The per-function instances are passed to later stages for variable recovery. By keeping these records separate, DESAN can use the same identification pass for reporting and as the front end of the optimization pipeline.

Core check identification is intentionally conservative. A call is forwarded only when its sanitizer family and check kind are recognized. For ASan, this includes load and store report calls such as `__asan_report_load*` and callback-style checks such as `__asan_load*`. For UBSan, it includes common handlers for type mismatch, pointer overflow, out-of-bounds access, and shift-out-of-bounds checks. For MSan, it includes warning, parameter, return-value, variadic-argument, and explicit shadow-query checks. Unrecognized sanitizer calls remain in the IR and are not candidates for deletion.

Checked variable identification. For each core check, DESAN recovers a checked variable by tracing the checked operand backward through the IR. The recovered object is represented as a tuple

$\langle \text{Base, Offsets, AccessType, Sanitizer, CheckInst} \rangle$.

Base is the normalized root object. *Offsets* describe the access path when it is recoverable. *AccessType* is READ, WRITE, or UNKNOWN. *Sanitizer* records the sanitizer family, and *CheckInst* points to the original sanitizer check.

The recovered variable identifies a candidate relationship between checks. It does not establish complete redundancy. Coverage additionally requires matching the checked property and its relevant operands or parameters. For ASan, these

include the accessed address range and access size. For UBSan, they depend on the predicate being checked, such as alignment, bounds, or pointer arithmetic. For MSan, coverage concerns the initialization state of the particular value or memory range being tested. A shared source-level name or sanitizer family does not establish equivalence.

For example, a UBSan `type_mismatch` check on `p` cannot cover a `pointer_overflow` check on `p+i`. The checks validate different properties, even when both are associated with subsequent reads. Likewise, two instances of the same handler are not necessarily equivalent if their operands or property-relevant parameters differ.

The tracing logic is sanitizer-specific at the entry point but shared once the checked value is found. For ASan report calls, the first argument is the checked address. In common inline ASan instrumentation, this address is produced before the shadow-memory computation and then passed to a report call on the slow path. DESAN traces this value backward through instructions such as `ptrtoint`, `inttoptr`, `bitcast`, and `getelementptr`. If the address originates from a field access, DESAN normalizes the variable to the base object and access path.

UBSan and MSan require different rules. For UBSan type mismatch checks, DESAN uses the pointer operand guarded by the handler. For pointer overflow, it considers the base and result pointer operands and prefers the recovered pointer value when possible. For out-of-bounds and shift checks, it traces the index or value operand when the checked use can be associated with nearby IR. For MSan, explicit value checks take the runtime-call argument directly. Warning-style checks require DESAN to recover the value from the surrounding shadow logic. If the checked operand cannot be recovered reliably, DESAN preserves the check.

DESAN also infers whether a check is READ, WRITE, or UNKNOWN. ASan load checks are READ, and ASan store checks are WRITE by construction. For UBSan and MSan, DESAN classifies checks from local IR uses. A checked pointer used by a load is READ; a checked pointer used as a store

destination is WRITE. Ambiguous cases are UNKNOWN. UNKNOWN checks are not removed. This policy prevents DESAN from treating complex undefined-behavior or shadow-state checks as ordinary read checks unless the IR context supports that interpretation.

Per-variable check CFG construction. After variable recovery, DESAN groups checks by normalized variable identity to organize candidate comparisons. This grouping is distinct from the equivalence required for elimination. Sharing a base object or sanitizer family does not show that two checks validate the same property. For example, accesses to $a[i]$ and $a[j]$ may belong to the same variable group but refer to different memory ranges. A coverage decision must establish that the checked property, object or value, and relevant access parameters match.

For each variable group, DESAN builds a per-variable check graph. Each node represents one sanitizer check. It records the check instruction and its metadata, including the access type, sanitizer kind, checked variable, basic block, and anchor instruction. For report-style ASan checks, the anchor is often the conditional branch guarding the report block, not the report call itself, because that branch is the control-flow point on the hot path. Edges represent reachability between checks in the original CFG. An edge records reachability between check points, respecting instruction order within a basic block and CFG paths across basic blocks.

5.2 Removed Checks

DESAN preserves WRITE and UNKNOWN checks. A READ check is eligible for elimination only if it satisfies the property-specific coverage condition defined below.

For a candidate READ check c , coverage requires an earlier READ or WRITE check on every execution path to c . That earlier check must successfully validate the same sanitizer property for the same object or value with matching property-relevant parameters. Its validation must remain valid until c . Different incoming paths may be covered by different checks. A single dominating check is sufficient, but coverage does not require one.

Analysis state. We formulate coverage as a forward must-analysis over the intraprocedural CFG. An abstract fact q identifies a checked property, its normalized object or value, and any property-relevant parameters. At each program point, the state S contains facts established on every path reaching that point. READ/WRITE classification describes a check rather than substituting for the identity of q .

For a reachable node B , the incoming state is

$$\text{IN}[B] = \begin{cases} \emptyset, & B \text{ is the function entry,} \\ \bigcap_{P \in \text{Pred}(B)} \text{OUT}[P], & \text{otherwise.} \end{cases}$$

Only reachable predecessors participate in the intersection. Consequently, a property may be available after a merge only

if every incoming path establishes it, even when those paths contain different covering checks.

Instructions update the state in execution order by removing potentially invalidated facts and adding facts on proven-success continuations. The guarded operation is processed separately from its check. A WRITE check may establish a fact, but a subsequent store can invalidate that fact before a later READ occurs. Original program instructions must remain in the transfer analysis, even if the check graph omits them. UNKNOWN checks are retained but do not generate coverage facts.

Coverage requires the checked property to remain valid. An unchanged variable name or pointer address alone is insufficient. An event invalidates a coverage fact if it may change the checked operands or the state on which the property depends. Such events include pointer reassignment, potentially aliasing writes, changes to object lifetime, and mutations of sanitizer metadata. Calls with unresolved effects must also invalidate any facts they may affect. If those facts cannot be identified conservatively, coverage must be discarded. Invalidation is property-specific. A store may change a value or its initialization state without changing the addressability of the containing storage. However, freeing an object invalidates addressability even if the pointer value does not change. A prior fact may survive an operation only if that operation is known to be irrelevant to the fact. Otherwise, the subsequent READ check must be retained.

Concurrent interference. Intraprocedural control-flow coverage alone does not establish stability against other threads. Synchronization operations, atomics, and thread-runtime calls require conservative treatment when they may expose changes to a checked property. However, treating these operations as local invalidation boundaries is not a proof that no other thread can modify the object or its sanitizer state between two checks. The preservation argument requires the absence of relevant concurrent interference. Its scope is discussed in Section 8.

Fixed point and candidate selection. Let Q be the finite set of candidate facts. We initialize non-entry states to Q and compute the following fixed point. The entry input remains empty, preventing a backedge from establishing coverage for an unchecked first iteration.

Algorithm 1 operates on the analysis CFG with logical check-success nodes. GEN adds facts only at recognized successful READ/WRITE validations, while KILL removes facts potentially invalidated by an event. Guarded operations are processed separately from their checks. The algorithm marks candidates without modifying the IR.

Figure 5 summarizes the main coverage cases. In the linear case, a previous READ or WRITE check directly covers a later READ for the same variable. In the branch fan-out case, one prior check covers multiple reads on outgoing branch paths. At a merge point, a READ can be removed even without a single dominating check if all incoming paths have already

Algorithm 1 Checked-property coverage analysis

Require: Reachable analysis CFG G ; candidate facts Q

Ensure: Removal candidates R

```
1:  $\text{OUT}[B] \leftarrow Q$  for each node  $B$ 
2: repeat
3:   for each node  $B$  do
4:     if  $B$  is the entry node then
5:        $\text{IN}[B] \leftarrow \emptyset$ 
6:     else
7:        $\text{IN}[B] \leftarrow \bigcap_{P \in \text{Pred}(B)} \text{OUT}[P]$ 
8:     end if
9:      $S \leftarrow \text{IN}[B]$ 
10:    for each event  $E$  in  $B$ , in execution order do
11:       $S \leftarrow (S \setminus \text{KILL}(E)) \cup \text{GEN}(E)$ 
12:    end for
13:     $\text{OUT}[B] \leftarrow S$ 
14:  end for
15: until no OUT state changes
16:  $R \leftarrow \emptyset$ 
17: for each node  $B$  do
18:    $S \leftarrow \text{IN}[B]$ 
19:   for each event  $E$  in  $B$ , in execution order do
20:     if  $E$  begins a READ check  $c$  and  $\text{fact}(c) \in S$  then
21:        $R \leftarrow R \cup \{c\}$ 
22:     end if
23:      $S \leftarrow (S \setminus \text{KILL}(E)) \cup \text{GEN}(E)$ 
24:   end for
25: end for
26: return  $R$ 
```

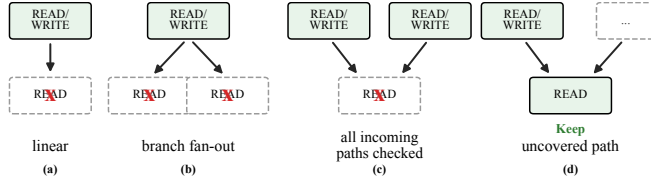


Figure 5. Coverage cases for redundant READ check elimination. Dashed nodes in (a)–(c) are removable READ checks, while the solid READ in case (d) is kept because one incoming path is not covered. All covering checks must validate the same property with matching parameters, and coverage must remain valid along the connecting paths.

checked the same variable. However, if one incoming path reaches the READ without a prior check, the READ is not covered and must be preserved.

Coverage requires a matching validation to remain valid on every reaching path. Neither instruction order nor preservation of WRITE checks alone establishes this condition. The analysis passes marked candidates and their check metadata to the slice remover without modifying the CFG.

5.3 Optimized Binary

The final stage converts the analysis result into optimized IR and then into a binary. This stage does more than erase a call instruction. Sanitizer checks, especially ASan checks, often

include several auxiliary instructions on the hot path. Removing only the report call would leave the shadow-memory load, comparison, and branch in place, so most of the runtime overhead would remain. Check elimination requires two separate conditions. The coverage analysis must establish that the check is redundant, and the transformation must preserve the original program’s computations and effects.

Check removal and IR cleanup. The slice removal starts from a redundant check call and collects a backward slice of sanitizer-only instructions. This slice may include address casts, shadow-address arithmetic, shadow-memory loads, comparisons, and conditional branches; it may also include empty inline assembly in report blocks and unreachable terminators. For ASan inline checks, DeSan follows the control-flow edge to the report block and tries to bypass or delete the block when it is isolated from original program logic. Control-flow rewrites must preserve the original continuation and update affected PHI operands. Shared computations are retained. Supporting instructions are deleted only if their removal has no required effects and leaves no remaining uses. If these conditions cannot be established, the candidate is preserved.

The remover applies a conservative safety filter. It may delete pure arithmetic, casts, comparisons, non-volatile and non-atomic shadow loads, and sanitizer-only control-flow instructions when their users are inside the slice. It leaves original program stores, atomics, allocas, volatile accesses, and instructions with unknown side effects intact. If DeSan cannot safely remove the full slice, it uses a smaller transformation instead, such as bypassing the report path or erasing a direct callback-style check call when that is safe.

After removing a check slice, DeSan runs local cleanup repeatedly. The cleanup pass deletes trivially dead instructions and unreachable blocks, and it merges basic blocks with unique predecessors when possible. These steps matter because sanitizer instrumentation often introduces small report blocks and control-flow diamonds. Once a redundant check is removed, standard CFG simplification can collapse those diamonds and expose further dead code.

Recompile to binary. The cleaned IR is recompiled with the original sanitizer runtime. DeSan does not require a custom runtime and does not change the ABI of sanitizer-instrumented binaries. In practice, the user compiles the source code with `clang -fsanitize=... -S -emit-llvm`, runs DeSan through `opt`, and invokes `clang` on the optimized IR with the same option so that the proper runtime is linked.

Because DeSan removes IR before final code generation, the standard compiler backend can further optimize the transformed program. Dead branches may disappear, register allocation may improve, and the resulting binary contains fewer sanitizer checks on covered read paths. Preserved WRITE, UNKNOWN, and uncovered READ checks still use the original sanitizer runtime, so existing sanitizer diagnostics and deployment workflows remain compatible.

6 Implementation

We implement DESAN as a loadable LLVM pass plugin using the new pass manager. The core implementation spans 13 C++ source/header files: about 3.8K physical lines of code. Table 2 summarizes the major components.

Table 2. Major DESAN components and their responsibilities.

Component	Main task
SanitizerCheckCollector	Classify sanitizer runtime calls
CheckedVariableAnalyzer	Trace checked operands backward
CheckGraphBuilder	Group checks by variable and CFG relation
RedundantCheckEliminator	Find covered READ checks
CheckSliceRemover	Delete sanitizer-only IR and simplify CFG

Sanitizer-specific analysis. The collector recognizes sanitizer calls primarily through stable runtime symbol names. ASan load/store checks, UBSan handler calls, and MSan warnings are mapped to sanitizer families and check types. Checked-variable recovery uses a shared backward tracer with sanitizer-specific entry rules. For ASan, the first report or callback argument is treated as the checked address and traced backward through casts, `ptrtoint/inttoptr`, shadow-address arithmetic, and `getelementptr`. For UBSan, the handler type determines the checked operand; examples include type mismatch, pointer overflow, bounds, and shift checks. For MSan, explicit value checks use runtime-call arguments, while warning-style checks require recovering nearby shadow-state operands. Access-type inference is conservative. ASan check names directly determine READ or WRITE. UBSan and MSan checks are classified using local IR uses when possible; otherwise they are marked UNKNOWN. UNKNOWN checks are retained by the eliminator.

Graph construction and elimination. Recovered checks are grouped by a normalized variable key containing the sanitizer family, base object, and available offsets. This key identifies candidates for comparison rather than establishing coverage: checks sharing a base may access different ranges, and elimination additionally requires matching checked properties and relevant parameters, as defined in Section 5.2. Each graph node records the check instruction, access type, basic block, and check anchor. For ASan report-style checks, the anchor is typically the conditional branch guarding the report block, which represents the hot-path validation point rather than the cold reporting call. CFG traversal and LLVM’s DominatorTree provide reachability and dominance information. A matching dominating check can establish coverage when no intervening event invalidates its property. Coverage provided by different incoming paths is established through the must-analysis in Section 5.2. The analysis accounts for intervening instructions and marks a READ check for removal only after convergence confirms valid coverage on every reaching path. WRITE and UNKNOWN checks remain unchanged.

IR transformation. CheckSliceRemover applies the transformation conditions in Section 5.3, retaining shared computations and original program effects. It removes eligible sanitizer-only instructions, then applies dead-code elimination and CFG simplification. Preserving a side-effecting instruction during cleanup does not imply that coverage remains valid across that instruction.

7 Evaluation

This section evaluates DESAN by asking how many sanitizer checks it removes, how much overhead it reduces, how it compares with existing sanitizer debloating systems, and whether it preserves the bug-detection behavior of the original sanitizers. We answer the following research questions.

RQ1: Effectiveness. How many sanitizer checks can DESAN remove, and how much sanitizer overhead does it reduce?

RQ2: Comparison. How does DESAN compare with existing sanitizer-check reduction systems?

RQ3: Detection capability. Does DESAN preserve the bug-detection capability of the original sanitizers?

7.1 Experimental Setup

Platform. All experiments were conducted on an x86_64 machine running Ubuntu 26.04 LTS with Linux kernel 7.0.0-15-generic. The machine was equipped with an Intel Core i9-8950HK CPU at 2.90GHz, 6 physical cores, 12 hardware threads, and 30GiB of main memory. We used LLVM/Clang 18.1.8 to build and evaluate DESAN, together with `clang-18`, `clang++-18`, `opt-18`, and `llvm-config-18`. For the comparison experiments, ASAN— and SanRazor were built and run with their required toolchains. All runtime measurements were taken on the same machine.

Datasets. We evaluate DESAN on SPEC CPU2006, SPEC CPU2017, and 20 widely used open-source projects. The open-source set consists of `bzip2`, `capstone`, `coreutils`, `curl`, `duktape`, `flac`, `grep`, `gzip`, `imagemagick`, `libpng`, `libvpx`, `lz4`, `make`, `openssl`, `redis`, `tar`, `xz`, `zlib`, `zopfli`, and `zstd`. For each program, when supported, we build native binaries, sanitizer-instrumented binaries, and binaries optimized by DESAN.

Filtering policy. We exclude a result from the aggregate statistics if any execution status is not successful, if the inserted check count is zero, or if the removed check count is zero. The resulting summaries are conditional on successful execution and nonzero check removal. After filtering, the evaluation includes 163 valid sanitizer configurations: 50 from SPEC CPU2006, 54 from SPEC CPU2017, and 59 from open-source projects. The detailed data is shown in Table 3. **Metrics.** For check reduction, we report *Core/All*, the ratio of core checks to all inserted sanitizer checks, and *Removed/All*, the ratio of removed checks to all inserted checks. For runtime, let *A* be the native runtime, *B* be the runtime with the original sanitizer, and *C* be the runtime after applying

Table 3. Evaluation datasets after filtering unsuccessful or zero-check configurations.

Dataset	Programs	ASan	UBSan	MSan
SPEC CPU2006	19	19	19	12
SPEC CPU2017	20	20	20	14
Open source	20	20	20	19
Total	59	59	59	45

DeSAN. We compute:

$$\text{Sanitizer Overhead} = \frac{B - A}{A} \times 100\%,$$

$$\text{DeSAN Overhead} = \frac{C - A}{A} \times 100\%,$$

$$\text{Overhead Reduction} = \frac{B - C}{B - A} \times 100\%.$$

We first compute each metric separately for every benchmark configuration using the corresponding runtimes A , B , and C . We then report the median for each metric across the configurations included in the comparison. Sanitizer Overhead, DeSAN Overhead, and Overhead Reduction are aggregated independently. In particular, the median Overhead Reduction is not calculated from the two median overhead values. We use the same procedure for baseline comparisons.

7.2 DeSAN Results

Table 4 summarizes DeSAN’s results. Across all valid configurations, the median Core/All ratio is 96.62%, showing that the core checks selected in Section 4 cover most sanitizer instrumentation. The median Removed/All ratio is 51.48%, which means DeSAN removes more than half of all inserted sanitizer checks among the retained configurations, while preserving WRITE and UNKNOWN checks. The CPU2017 MSan row illustrates the distinction between per-configuration metrics and their independently computed medians. Its median DeSAN Overhead is 364.33%, compared with a median Sanitizer Overhead of 273.33%. Its Overhead Reduction is 34.04%.

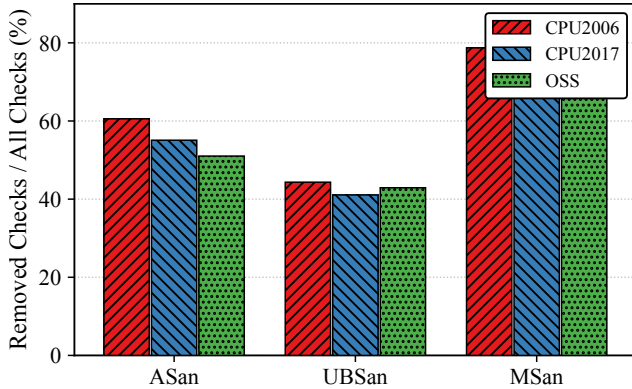


Figure 6. DeSAN removed-check ratio. Bars report medians over valid benchmarks.

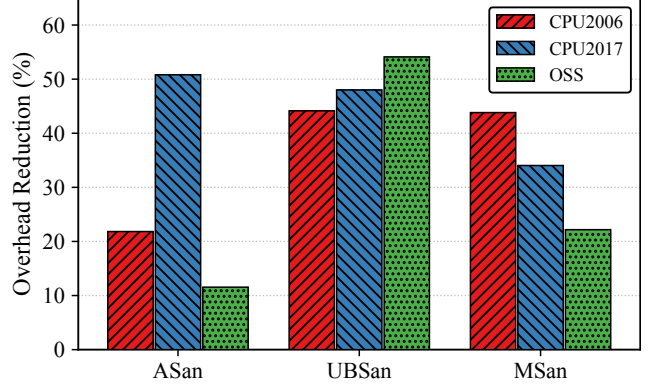


Figure 7. DeSAN sanitizer-overhead reduction. Each bar reports the median of per-configuration overhead reductions over the retained configurations.

Figures 6 and 7 show the static and runtime effects. MSan has the highest static removal ratio because many MSan checks repeatedly validate propagated shadow state. DeSAN removes 78.72% of all checks on CPU2006, 78.38% on CPU2017, and 72.52% on open-source projects. ASan also shows substantial redundancy, with median Removed/All ratios of 60.56%, 55.08%, and 51.02%. UBSan has lower static removal ratios because its handlers are more semantically diverse and more often classified conservatively, but it still achieves strong runtime reductions.

Runtime improvement depends on where the removed checks execute and how expensive they are at runtime. On CPU2017, DeSAN reduces ASan overhead by 50.81% at the median; on open-source projects, it reduces UBSan overhead by 54.10%. However, reducing the static check count does not translate linearly into runtime speedup. The runtime benefit of removing a check depends on its execution frequency and per-check cost, and sanitizer families attach different runtime costs to their checks. For ASan, deleting the full check slice can remove the hot-path shadow work along with the associated shadow-memory access and branch logic. For UBSan and MSan, the cost may depend more on how the guarded value is used and how shadow-state propagates. Therefore, some configurations show lower runtime improvement despite high Removed/All ratios, especially when the removed checks are dynamically cold, the original sanitizer overhead is small, or execution time is dominated by I/O or library calls. This is why we report both Removed/All and Overhead Reduction rather than relying on static check count alone.

7.3 Comparison with Existing Work

We compare DeSAN with two existing systems. ASAN-- [36] is a state-of-the-art ASan optimization tool, so we use it as the ASan baseline. SanRazor [35] can target ASan and UBSan, but its ASan performance is weaker than ASAN-- in our setting; for this reason, we compare SanRazor with DeSAN on UBSan. SanRazor relies heavily on test workloads

Table 4. DeSAN results on SPEC CPU2006, SPEC CPU2017, and open-source projects. N is the number of retained configurations. All other columns report independently computed medians. Overhead Reduction is calculated for each configuration as $(B - C)/(B - A) \times 100\%$ before aggregation. It is not derived from the median overhead columns.

Dataset	Sanitizer	N	Core/All	Removed/All	Sanitizer Overhead	DeSAN Overhead	Overhead Reduction
CPU2006	ASan	19	94.05%	60.56%	104.02%	62.88%	21.83%
CPU2006	UBSan	19	84.84%	44.34%	218.70%	128.10%	44.14%
CPU2006	MSan	12	100.00%	78.72%	471.93%	254.95%	43.82%
CPU2017	ASan	20	95.86%	55.08%	102.45%	44.20%	50.81%
CPU2017	UBSan	20	87.31%	41.07%	314.42%	183.90%	48.00%
CPU2017	MSan	14	100.00%	78.38%	273.33%	364.33%	34.04%
Open source	ASan	20	94.89%	51.02%	109.65%	88.55%	11.55%
Open source	UBSan	20	89.08%	42.91%	88.00%	43.93%	54.10%
Open source	MSan	19	100.00%	72.52%	254.24%	155.04%	22.18%

to decide which checks can be disabled. SPEC benchmarks are favorable for SanRazor because they provide standardized workloads and repeatable executions. However, many open-source projects are missing test suites, have incomplete test suites, or have tests that miss performance-critical paths. This limits the practical use of a workload-dependent reduction strategy. For SanRazor, we treat *Detected Disabled* checks as removed checks.

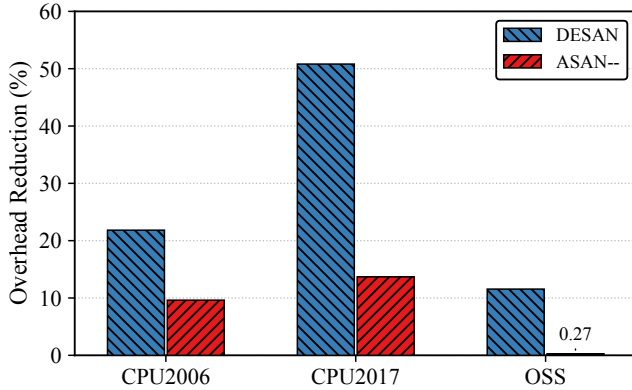


Figure 8. Comparison with ASAN-- on ASan. Each bar reports the median of per-configuration overhead reductions over the common successful benchmarks.

Figure 8 summarizes the ASan runtime comparison with ASAN--. Figure 9 shows the semantic composition of checks removed by SanRazor and DeSAN on UBSan; Table 6 reports the corresponding overhead-reduction values. This comparison measures check reduction, runtime overhead, and the access-type composition of removed checks. The composition does not establish detection losses. UNKNOWN denotes unresolved access semantics under our classification, rather than an absence of justification in the baseline.

Table 5 shows that DeSAN removes more ASan checks than ASAN-- on all three datasets and achieves a higher median overhead reduction on common successful benchmarks. ASAN-- removes both READ and WRITE checks.

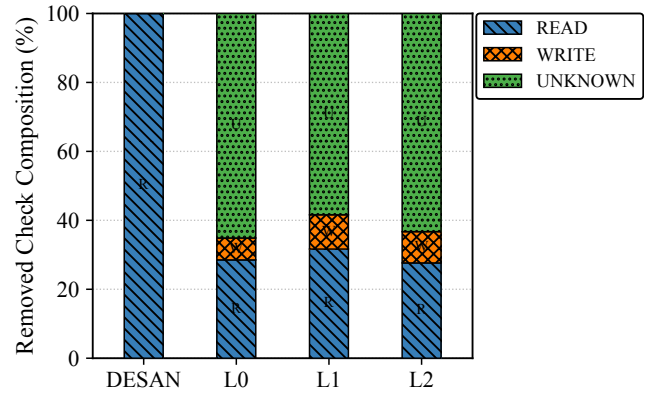


Figure 9. Composition of checks removed by DeSAN and SanRazor on UBSan for SPEC CPU2006. The composition is computed from total removed checks across common successful benchmarks. SanRazor removes READ, WRITE, and UNKNOWN checks, whereas DeSAN removes only READ.

In our measurements, WRITE checks make up a median of 30.95% of ASAN--’s removed checks on CPU2006, 36.49% on CPU2017, and 34.12% on open-source projects. DeSAN preserves WRITE checks under its removal policy and obtains its reduction from READ-check elimination and sanitizer-slice cleanup. The open-source result for ASAN-- does not mean that ASAN-- removed zero checks. It still removes a median of 25.43% of checks, but it reduces median runtime overhead by only 0.27%, which appears close to zero in Figure 8. This gap may reflect cold removed checks or workloads dominated by I/O and library execution; identifying the dominant cause requires further profiling.

Table 6 shows a different tradeoff for UBSan. On the common CPU2006 subset, SanRazor-L2 removes more checks than DeSAN and achieves a higher median overhead reduction. Across SanRazor levels, more than half of the detected disabled checks are classified as UNKNOWN, and 5.26%–7.10% are classified as WRITE. DeSAN restricts elimination to READ checks and retains WRITE and UNKNOWN checks.

Table 5. Comparison with ASAN-- on ASan. All values are medians over common successful benchmarks. DeSAN removes only READ checks; ASAN-- removes both READ and WRITE checks. READ and WRITE percentages describe the composition of removed checks, using all removed checks as the denominator. The overall removal percentage uses all inserted checks as the denominator. (OH: Overhead)

Dataset	N	DeSAN Removed	ASAN-- Removed	ASAN-- READ	ASAN-- WRITE	DeSAN OH Reduced	ASAN-- OH Reduced
CPU2006	19	60.56%	30.45%	69.05%	30.95%	21.83%	9.62%
CPU2017	20	55.08%	30.95%	63.51%	36.49%	50.81%	13.68%
Open source	20	51.02%	25.43%	65.88%	34.12%	11.55%	0.27%

Table 6. Comparison with SanRazor on UBSan for SPEC CPU2006. SanRazor removed checks are counted using Detected Disabled. All values are medians over common successful benchmarks. READ, WRITE, and UNKNOWN percentages use all removed checks as the denominator (Detected Disabled for SanRazor), whereas the overall removal percentage uses all inserted checks.

Method	N	Removed	Overhead Reduction	Removed READ	Removed WRITE	Removed UNKNOWN
DeSAN	10	46.34%	47.91%	100.00%	0.00%	0.00%
SanRazor-L0	10	40.31%	12.62%	26.79%	5.26%	68.52%
SanRazor-L1	10	51.44%	30.55%	28.76%	7.10%	57.78%
SanRazor-L2	10	70.65%	69.08%	26.68%	6.91%	63.48%

7.4 Vulnerability Detection

We evaluate detection preservation by comparing DeSAN-optimized binaries with binaries instrumented by the corresponding original sanitizer on the same bug-triggering inputs. The Juliet Test Suite (v1.3) [20] evaluation includes 7,460 cases detectable by the original sanitizers. ASan detects 4,546 cases, UBSan 2,148, and MSan 766. As shown in Table 7, all of these cases remain detectable after applying DeSAN. This represents 100% retention of baseline-detectable cases.

Table 7. Detection retention on the evaluated Juliet cases. All listed cases are detectable by the original sanitizer. Percentages report retention after DeSAN optimization.

Description	CWE	Total	Detected By	DeSAN
Stack Buffer Overflow	121	2348	ASan	2348 (100%)
Heap Buffer Overflow	122	1677	ASan	1677 (100%)
Double Free	415	362	ASan	362 (100%)
Use After Free	416	159	ASan	159 (100%)
Integer Overflow	190	1536	UBSan	1536 (100%)
Null Pointer Dereference	476	348	UBSan	348 (100%)
Divide By Zero	369	264	UBSan	264 (100%)
Uninitialized Variable	457	766	MSan	766 (100%)

We also collect 108 real-world vulnerabilities detectable by ASan, UBSan, and MSan. We select the dataset using two criteria. First, the vulnerable programs come from an actively maintained CVE list [5, 18], which includes detailed instructions for compiling each program and reproducing the vulnerabilities. Second, each selected program must run with the provided input data so that sanitizers can detect the triggered errors. As shown in Table 8, all real-world vulnerabilities remain detectable after applying DeSAN.

Table 8. Results of real-world vulnerability detection.

Sanitizer	Vulnerability Category	Total	DeSAN Detectable
ASan	Heap overflow	43	43 (100%)
ASan	Stack overflow	24	24 (100%)
ASan	Use after free	3	3 (100%)
UBSan	Integer overflow	21	21 (100%)
UBSan	Null pointer dereference	15	15 (100%)
MSan	Uninitialized memory use	2	2 (100%)
Total	–	108	108 (100%)

All 108 evaluated real-world vulnerabilities detected by the original sanitizers remain detectable after DeSan optimization. Together with the Juliet results, this provides empirical evidence of detection preservation for the evaluated programs and inputs. This detection evaluation compares DeSAN with the original sanitizers. ASAN-- and SanRazor were not evaluated on the same complete bug corpus, so these results do not establish comparative detection rates for the baselines.

8 Discussion

Scope. DeSAN targets redundant READ checks in sanitizer-instrumented LLVM IR for ASan, UBSan, and MSan while retaining the original sanitizer runtime. Its preservation argument is conditional on the property-specific coverage and invalidation assumptions in Section 5.2, including the concurrency limitations discussed below.

Concurrency. DeSAN’s coverage argument is intra-thread and assumes no other thread changes the relevant property between the covering validation and the candidate READ. Consider a thread that checks and accesses an object, after which another thread frees the object before the first thread

accesses it again. The earlier validation no longer shows that the object is addressable. A later READ check may detect the resulting violation, which would be missed if the check were removed. An intra-thread CFG alone cannot exclude this execution. Nor does local invalidation at synchronization or atomic operations establish the required absence of interference. Therefore, our preservation argument does not generally cover shared objects that other threads may modify. Extending the guarantee requires reasoning about how such objects are shared, synchronized, and kept alive.

Performance interpretation. Section 7 reports static reduction and runtime overhead separately because check frequency and cost differ. Small baseline overhead makes the overhead-reduction ratio particularly sensitive to timing variation. Code-layout and backend effects may also influence runtime, but attributing individual regressions to these mechanisms would require additional measurements.

Design boundaries. DESAN deliberately favors conservative, explainable transformations. The pass preserves a check when it cannot recover the checked variables, when the access type is UNKNOWN, or when aliasing, pointer re-assignment, or unclear interprocedural effects may break equivalence. The pass also leaves non-core checks and wrapper functions untouched, since they may do more than pure checking. These boundaries reduce the number of checks that can be removed, but they keep the optimization focused on redundant-check elimination rather than on rewriting sanitizer semantics. Still, the design leaves room for extension without changing DESAN’s core safety policy. Stronger interprocedural recovery and sanitizer-specific modeling could allow more UNKNOWN checks to be reclassified as removable READ checks.

9 Related Work

Sanitizer systems. Compiler-based sanitizers are widely used to detect memory errors and other undefined behavior. AddressSanitizer (ASan) uses shadow memory and redzones to detect out-of-bounds accesses and use-after-free errors [14, 24]. UndefinedBehaviorSanitizer (UBSan) inserts checks for cases such as misaligned accesses, invalid type uses, pointer overflow, and integer errors [16]. MemorySanitizer (MSan) tracks uninitialized values through shadow state [15]. Other sanitizer designs and memory-safety systems reduce deployment costs by changing how checks are performed, sampling checks, using hardware support, or focusing on particular languages and domains [1, 3, 7, 12, 17, 19, 25, 26]. These systems mainly introduce new detection mechanisms or runtime policies. DESAN takes a complementary path: it does not add a new sanitizer runtime, but removes redundant checks after existing sanitizers have inserted their instrumentation [31].

Dynamic and profile-guided check reduction. Dynamic sanitizer check reduction uses runtime information to find

checks that appear unnecessary in observed executions [8, 11, 13, 22, 32]. ASAP [29] uses profiling to reduce the cost of expensive sanity checks under an overhead budget. SanRazor [35] uses workload coverage together with static dependency analysis to disable sanitizer checks that appear redundant on profiled executions. Profiling reveals which checks execute frequently and whether their outcomes consistently agree across observed executions, helping prioritize costly checks and identify potentially redundant ones. Such agreement, however, does not establish equivalence on unobserved executions. These approaches therefore depend on representative profiling inputs. Standardized benchmark workloads provide a controlled setting for evaluating such methods, while incomplete application tests may yield less representative profiles. DESAN does not require profiling inputs to make elimination decisions. It expresses coverage through checked operands, properties, and control flow.

Static sanitizer optimization. Static sanitizer optimization reduces instrumentation cost at compile time. ASan— [36] applies static debloating techniques to reduce ASan overhead, and other work studies compiler/sanitizer interactions, sanitizer-eliding optimizations, and specialized pruning for particular sanitizer families [2, 6, 10, 21, 33, 34]. DESAN separates its removal policy from its coverage requirements. It retains WRITE and UNKNOWN checks, while removing candidate READ checks requires reasoning about the protected property and any intervening effects. The framework also removes associated sanitizer-only instructions. A no-WRITE-removal filter restricts an optimizer’s candidate set but does not recover checked operands, establish property equivalence, or account for invalidation.

10 Conclusion

DESAN explores a conservative approach to sanitizer-check reduction. It restricts elimination to READ checks and distinguishes this policy from the property-specific conditions that establish redundancy. Its design recovers checked operands and uses coverage reasoning to remove sanitizer-only slices. Across 163 retained configurations from SPEC CPU2006, SPEC CPU2017, and 20 open-source projects, the prototype removes a median of 51.48% of inserted checks. It also achieves competitive overhead reductions compared with ASan— and SanRazor. After optimization, all 7,460 evaluated Juliet cases and 108 real-world vulnerabilities detectable by the original sanitizers remain detectable.

Acknowledgments

We thank our shepherd and the reviewers for their insightful feedback. This work is partially supported by the U.S. Office of Naval Research grant N00014-23-1-2122, NSF grants 2623086 and 2450603, CCI AI for Cyber-security grant N-3Q25-002, and a gift from VISA Inc.

References

- [1] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. 2023. {MTSan}: A Feasible and Practical Memory Sanitizer for Fuzzing {COTS} Binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*. 841–858.
- [2] Yanzuo Chen, Yuanyuan Yuan, and Shuai Wang. 2023. OBSan: An Out-Of-Bound Sanitizer to Harden DNN Executables. In *NDSS*.
- [3] Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee. 2024. {RustSan}: Retrofitting {AddressSanitizer} for Efficient Sanitization of Rust. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3729–3746.
- [4] Dominik Klemba, Dominik Czarnota. 2024. Understanding AddressSanitizer: Better memory safety for your code. <https://blog.trailofbits.com/2024/05/16/understanding-addresssanitizer-better-memory-safety-for-your-code/>.
- [5] Dongliang Xu. 2019. CVE list. https://vulnreproduction.github.io/Primary_CVE/.
- [6] Cormac Flanagan and Stephen N Freund. 2013. Redcard: Redundant check elimination for dynamic race detectors. In *ECOOP 2013—Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*. Springer, 255–280.
- [7] Rahul George, Mingming Chen, Kaiming Huang, Zhiyun Qian, Thomas La Porta, and Trent Jaeger. 2024. {OPTISAN}: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget. In *33rd USENIX Security Symposium (USENIX Security 24)*.
- [8] Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient n-version execution framework. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 339–353.
- [9] HPCWiki. 2025. Compiler Sanitizer. https://hpc-wiki.info/hpc/Compiler_Sanitizers.
- [10] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik Van Der Kouwe, and Klaus von Gleissenthall. 2023. Don't look UB: Exposing sanitizer-eliding compiler optimizations. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 907–927.
- [11] Anil Kurmus and Robby Zippel. 2014. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1366–1377.
- [12] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 721–732.
- [13] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. 2018. PartiSan: fast and flexible sanitization via run-time partitioning. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer, 403–422.
- [14] LLVM Clang. 2024. Address Sanitizer. <https://clang.llvm.org/docs/AddressSanitizer.html>.
- [15] LLVM Clang. 2024. Memory Sanitizer. <https://clang.llvm.org/docs/MemorySanitizer.html>.
- [16] LLVM Clang. 2024. Undefined Behavior Sanitizer. <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>.
- [17] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. ERASAN: Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society.
- [18] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 919–936.
- [19] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258.
- [20] NIST. 2017. Software assurance reference dataset. <https://samate.nist.gov/SRD/testsuite.php>.
- [21] Dustin Rhodes, Cormac Flanagan, and Stephen N Freund. 2017. Bigfoot: Static check placement for dynamic race detection. *ACM SIGPLAN Notices* 52, 6 (2017), 141–156.
- [22] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and thanks for all the bugs: Finding errors in c programs by abstracting from the native execution model. *ACM SIGPLAN Notices* 53, 2 (2018), 377–391.
- [23] Kostya Serebryany. 2016. Sanitize, Fuzz, and Harden Your C++ Code. In *USENIX Enigma*. USENIX Association, San Francisco, CA.
- [24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.
- [25] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrlkevich, Christian Holler, Julian Lettner, et al. 2024. Gwp-asan: Sampling-based detection of memory-safety bugs in production. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*.
- [26] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrlkevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *arXiv preprint arXiv:1802.09517* (2018).
- [27] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1275–1295.
- [28] Emanuel Vintila, Philipp Zieris, and Julian Horsch. 2025. Evaluating the Effectiveness of Memory Safety Sanitizers. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society.
- [29] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 866–879.
- [30] Wikipedia. 2025. Code Sanitizer. https://en.wikipedia.org/wiki/Code_sanitizer.
- [31] Yunlong Xing. 2026. *Enhancing Memory Safety: From Efficient Sanitization to Context-Aware Patch Generation*. Ph.D. Dissertation. George Mason University.
- [32] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. 2017. Bunshin: compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 271–283.
- [33] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 2017. Simber: Eliminating redundant memory bound checks via statistical inference. In *ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings 32*. Springer, 413–426.
- [34] Yizhuo Zhai, Zhiyun Qian, Chengyu Song, Manu Sridharan, Trent Jaeger, Paul Yu, and Srikanth V Krishnamurthy. 2024. Don't Waste My Efforts: Pruning Redundant Sanitizer Checks by {Developer-Implemented} Type Checks. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1419–1434.
- [35] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. 2021. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 479–494.
- [36] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triantopoulos, and Jun Xu. 2022. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*. 4345–4363.