SanRazor: Reducing Redundant Sanitizer Checks in C/C++ Programs

Jiang Zhang[1] Shuai Wang[2*] Manuel Rigger[3] Pingjia He[3] Zhendong Su[3]

University of Southern California[1] *HKUST*[2] *ETH Zurich*[3]

Abstract

Sanitizers detect unsafe actions such as invalid memory accesses by inserting checks that are validated during a program's execution. Despite their extensive use for debugging and vulnerability discovery, sanitizer checks often induce a high runtime cost. One important reason for the high cost is, as we observe in this paper, that many sanitizer checks are *redundant* — the same safety property is repeatedly checked — leading to unnecessarily wasted computing resources.

To help more profitably utilize sanitizers, we introduce SanRazor, a practical tool aiming to effectively detect and remove redundant sanitizer checks. SanRazor adopts a novel hybrid approach — it captures both dynamic code coverage and static data dependencies of checks, and uses the extracted information to perform a redundant check analysis. Our evaluation on the SPEC benchmarks shows that SanRazor can reduce the overhead of sanitizers significantly, from 73.8% to 28.0–62.0% for AddressSanitizer, and from 160.1% to 36.6–124.4% for UndefinedBehaviorSanitizer (depending on the applied reduction scheme). Our further evaluation on 38 CVEs from 10 commonly-used programs shows that SanRazor-reduced checks suffice to detect at least 33 out of the 38 CVEs. Furthermore, by combining SanRazor with an existing sanitizer reduction tool ASAP, we show synergistic effect by reducing the runtime cost to only 7.0% with a reasonable tradeoff of security.

1 Introduction

Software *sanitizers* are designed to detect software bugs and vulnerabilities in code written in unsafe languages like C/C++ [33]. A sanitizer typically inserts additional checks into the program during compilation; at run time, the sanitizer check terminates the program if it detects unsafe actions or states (e.g., a buffer overflow). To date, various sanitizers have been designed to help detect vulnerabilities in C/C++ programs [4, 8, 24, 32, 35].

Sanitizers are commonly used by developers to find bugs before software deployment. In principle, they could also be used in deployed software, where they terminate program executions to prevent vulnerabilities from being exploited. In practice, however, the high runtime overhead of sanitizers inhibits their adoption in this application scenario [33, 40]. For example, our study on SPEC CPU2006 benchmarks [34] shows that the geometric mean overheads induced by AddressSanitizer (ASan) [32] and UndefinedBehaviorSanitizer (UBSan) [8] are, respectively, 73.8% and 160.1% (cf. Sec. 6).

It is difficult to reduce the overhead of sanitizer checks and therefore accelerate the execution of sanitization-enabled programs. To date, a number of approaches have been proposed aiming at finding unnecessary sanitizer checks with static analysis [6, 9, 11, 12, 15, 25, 37, 39, 44, 45]. For example, some approaches remove array bound checks by checking whether the value range of an index falls within the array size. They usually perform heavyweight, specialized program analyses to reduce specific sanitizer checks. In contrast, ASAP [40], the most closely related work to ours, elides sanitizer checks deemed the most costly based on a user-provided overhead budget. Despite being general and supporting sanitizers of different implementations, ASAP removes checks irrespective of their importance and may overoptimisitcally remove them, resulting in missed vulnerabilities. Thus, prior approaches for reducing sanitizer checks use either sanitizer-specific static analyses (e.g., [9, 37]) to remove only semantically redundant checks, or general heuristics [40] to remove costly sanitizer checks irrespective of their semantics.

This work explores a new, novel design point — it introduces a general framework, SanRazor, for effectively removing likely redundant checks. SanRazor is designed as a hybrid approach. First, it gathers coverage statistics during a profiling phase (e.g., based on a program's test suite). It then performs a correlation analysis, employing both the profiled coverage patterns as well as static data dependencies, to pinpoint and remove checks identified as likely redundant. Like ASAP [40], SanRazor is general and orthogonal to existing sanitizer reduction approaches that focus on specific

*Corresponding author.

sanitizer check implementations [9, 37, 44]. Distinct from ASAP, SanRazor identifies and removes sanitizer checks that repeatedly check the same program property, while ASAP removes sanitizer checks of high cost and may miss vulnerabilities. Although, like ASAP, SanRazor is *unsound*, i.e., it may remove checks even when they are unique, in practice, our evaluation results show that it accurately maintains the sanitizer's effectiveness in discovering defects and provides significantly reduced runtime overhead.

We evaluate the performance gain of SanRazor on the SPEC CPU2006 benchmark. The results show that SanRazor reduces geometric mean runtime overhead caused by ASan from 73.8% to 28.0–62.0% (depending on the different reduction schemes in SanRazor). Similarly, geometric mean overhead incurred by UBSan on SPEC programs is reduced from 160.1% to 36.6–124.4%. To measure the accuracy of SanRazor, we evaluate 10 popular programs with a total of 38 known CVEs. Results show that after removing redundant sanitizer checks, at least 33 CVEs can still be discovered. Compared with ASAP, SanRazor significantly outperforms ASAP by discovering more CVEs when achieving the same amount of cost reduction. We also explored practical methods to combine SanRazor and ASAP and reduce runtime cost to only 7.0% with a reasonable tradeoff of security. These promising results suggest that SanRazor could help promote the adoption of sanitizers in production usage. In sum, we make the following main contributions:

- At the conceptual level, we introduce the novel approach to reducing performance overhead incurred by sanitizers by identifying and removing likely redundant checks. By reducing sanitizer cost, sanitization-enabled programs can be executed faster, making sanitizer adoption in production use more practical.
- At the technical level, we design and implement a practical tool, SanRazor, to reduce sanitizer checks. SanRazor performs a hybrid analysis by leveraging both coverage patterns and static data dependency features to identify sanitizer checks as likely redundant.
- At the empirical level, our evaluation on the SPEC benchmarks shows that SanRazor can significantly reduce runtime overhead caused by ASan and UBSan. Moreover, our evaluation on real-world software with known CVEs shows that after applying SanRazor to reduce sanitizer checks, almost all CVEs can still be discovered.

We have publicly released SanRazor on GitHub at https://github.com/SanRazor-repo/SanRazor.

2 Preliminaries

Sanitizers are dynamic tools for finding software defects [33]. Sanitizers insert *sanitizer checks*, which are statements for monitoring program behaviors and validating whether they violate certain properties. We now introduce two sanitizers provided by the LLVM framework, ASan and UBSan, which have helped to detect many vulnerabilities [33].

ASan. Memory access errors like buffer overflow and use-after-free are severe vulnerabilities in C/C++ programs. ASan is designed to detect memory errors [32], and consists of an instrumentation module and a runtime library. The instrumentation module allocates shadow memory regions for each memory address used by the program. It also instruments each memory load and store operation such that before a memory address a is used to access memory, a will be mapped to its corresponding shadow memory address sa; the value stored in sa is then loaded and checked to decide whether the access via a is safe. The instrumentation module also allocates a "bad" region for each shadow memory region; directly using a shadow memory address sa in the application code will be redirected to the "bad" region, which is inaccessible via page protection. The runtime library hooks the `malloc` function to create poisoned "redzones" next to allocated memories to detect memory access errors. Similarly, the `free` function is instrumented to put the entire deallocated memory region into "redzones." This ensures that the recently-freed region will not be used by `malloc` for reallocation.

UBSan. Undefined behaviors can incur severe software vulnerabilities [41]. UBSan [8] detects a large set of common undefined behaviors in C/C++ code, such as out-of-bounds access, divided by zero, and invalid shift. We briefly introduce one undefined behavior that UBSan can detect:

Out-of-bounds Array Access Unlike ASan, which relies on shadow memory, UBSan detects out-of-bounds array accesses by comparing each array index with the array size. Consider the sample code below:

```
1 UChar buf[32];   // buf size is 32
2 for(i = 0; i < nBuf; i++)
3   out[i] = buf[nBuf-i-1];
```

where `buf` has 32 elements. When `nBuf` is greater than 32, executing `buf[nBuf-i-1]` may trigger an out-of-bounds access (e.g. when `i` is 0). UBSan identifies this by placing an extra `if` condition to compare the array index `nBuf-i-1` with the array size 32 before executing the loop body.

3 Problem Formulation

Conceptually, a sanitizer check $c(v)$ (v is the input parameter) can be defined as follows:

```
if(P(v) does not hold) abort_or_alert();
```

where c checks whether a property P holds w.r.t. parameter v. Usually, v denotes critical program information (e.g., code pointers), and by violating property P, e.g., a null pointer dereference, c either aborts program execution or alerts the user. Considering a program p with N sanitizer checks inserted, we use $c_i.v$ and $c_i.P$ to denote the parameter of the ith check c_i and its checked property throughout this section.

As introduced in Sec. 2, computation overhead can be introduced by each c_i, since c_i performs complex safety property checking, and may require extra memory to store metadata.

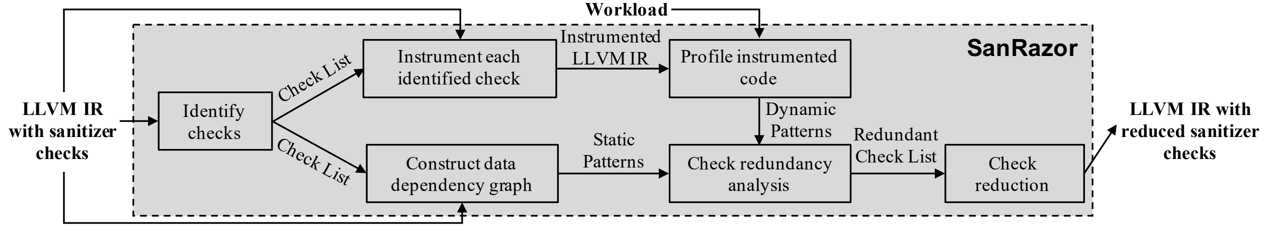

Figure 1: Workflow of SanRazor.

Nevertheless, a large portion of sanitizer checks repeatedly assert *identical* properties, thus wasting computing resources on properties deemed safe. We aim to remove redundant checks to reduce cost, thus making the production adoption of sanitizer checks more practically feasible. Next, we present a motivating example, and then formulate the notion of "redundant sanitizer checks" from a functionality perspective.

3.1 Sample Redundant Checks in bzip2

We show an example of checks in `bzip2` that repeatedly validate the same array index as follows:

```
for(i=0; i < nblock; i++) {
 j = eclass8[i]; //ASan1
 k = ftab[j] - 1; //ASan2;
 ftab[j] = k; //ASan3; ASan2 and ASan3 identical?
 fmap[k] = i; //ASan4; k < fmap_size always hold?
```

When ASan is enabled, four sanitizer checks are inserted to detect out-of-bound array access. Existing research could remove `ASan4`, by asserting `k` always falls within the size of `fmap`. In contrast, SanRazor advocates a new and orthogonal focus by deciding that `ASan2` and `ASan3` validate the same index, and therefore, `ASan3` can be removed without missing potential defects.[1] Indeed, our study shows that check redundancy is a general concern in real-world software (see Sec. 6), motivating a strong need for optimization. Also, to the best of our knowledge, standard compiler optimizations and previous research in this field (e.g., [6,9,11,12,15,28]) do not strive to use "similarity analysis" to reveal the equalivance of `ASan2` and `ASan3` and shave `ASan3` accordingly. In fact, our study shows that, when full optimizations (`-O3`) of `clang` are enabled, no sanitizers can be shaved for this case. This observation underlies the key novelty of SanRazor, whose design will be introduced in Sec. 4.

3.2 Redundant Sanitizer Checks

We start by giving a general definition of what a redundant check is, before refining the notion in an operational way:

Definition 1. *Assume that a sanitizer check c_i that could detect a hypothetical bug B in program p is removed. If B can still be detected, either by another sanitizer check c_j or by a user-defined check, then c_i is a redundant sanitizer check.*

[1] We scope SanRazor to single threaded programs. See Sec. 4 for further discussion of application scope.

More formally, given a nontrivial, single-threaded program p with a set of checks $c \in \mathbb{C}$, two checks c_i and c_j are deemed identical, when the following condition holds:

$$(c_i \in \text{dom}(c_j) \vee c_j \in \text{dom}(c_i)) \wedge [\![c_i.v]\!] = [\![c_j.v]\!] \wedge c_i.P = c_j.P$$

where $c_i \in \text{dom}(c_j)$ and $c_j \in \text{dom}(c_i)$ denote that c_i dominates c_j in the control flow graph or vice versa. Therefore, every execution from the program entry point to c_j goes through c_i or vice versa [5]. $[\![c_i.v]\!] = [\![c_j.v]\!]$ represents that $c_i.v$ and $c_j.v$ are semantically equivalent. $c_i.P = c_j.P$ means that c_i and c_j are the same kind of checks (e.g., they are both ASan checks, which can be recognized with pattern matching; cf. Sec. 4.1). When c_i and c_j satisfy the given condition and $c_i \in \text{dom}(c_j)$, c_j can be removed because if c_j is executed, c_i must be executed and they check the same property. The given condition specifies the functional equivalence of c_i and c_j. However, computability theory (e.g., Rice's theorem [29]) suggests that it could be very difficult, if possible at all, to assert $[\![c_i.v]\!] = [\![c_j.v]\!]$ for nontrivial programs. Moreover, performing control flow analysis to recover the dominator tree (e.g., $\text{dom}(c_j)$) information can be challenging and lead to false alarms, especially for cases where points-to analyses are extensively used in performing control flow analysis.

Given the theoretical challenge of identifying redundant sanitizer checks, we instead propose a practical approximation to identify *likely redundant checks*. Our approximation extracts both code coverage patterns and static input dependency patterns of checks (cf. Sec. 4); two checks are deemed "redundant" when they yield identical dynamic and static patterns. Specifically, we search for a pair of checks c_i and c_j and flag them as redundant if all the following conditions hold:

- c_i and c_j have correlated dynamic code coverage patterns, when executing the software with a nontrivial amount of workload inputs. Here, coverage patterns are checked regarding their "correlation", such that they can be identical, or one check's coverage pattern can subsume the other's pattern (see Sec. 4.4 for details).
- $c_i.P(c_i.v)$ and $c_j.P(c_j.v)$ are approximately equivalent w.r.t. static data dependency patterns deduced by our technique: $[\![c_i.P(c_i.v)]\!] \approx [\![c_j.P(c_j.v)]\!]$.

The first condition can be determined by instrumenting and profiling the program, and for the second, we assert

$[\![c_i.P(c_i.v)]\!] \approx [\![c_j.P(c_j.v)]\!]$ by checking the data dependency of two check inputs (see Sec. 4.4 for technical details).

4 Design

Fig. 1 depicts the workflow of SANRAZOR. SANRAZOR starts by identifying both user-defined checks and sanitizer checks (Sec. 4.1). It then instruments each check to record code coverage patterns (Sec. 4.2). We select a suitable workload (**Workload** in Fig. 1) and run the instrumented program with this workload to gather code coverage for each check. SANRAZOR then performs static analysis to construct data dependency graphs per check input and extract static patterns (Sec. 4.3). After obtaining static and dynamic characteristics for each check, SANRAZOR conducts an *unsound* check redundancy analysis (Sec. 4.4). The dynamic and static patterns are both analyzed, and checks with identical patterns will be marked as redundant and removed. The program with remaining checks will be compiled into an executable.

Application Scope. We implement SANRAZOR to analyze LLVM intermediate representation (IR) [17] and remove redundant ASan and UBSan checks. While the current implementation focuses on C/C++ programs, SANRAZOR does not rely on any specific features of C/C++. Therefore, programs written in any programming language can be analyzed, as long as they can be compiled to LLVM IR. Static and dynamic patterns leveraged by SANRAZOR are orthogonal to particular sanitizer implementations; hence, in principle SANRAZOR can reduce checks of different sanitizers. Contrarily, existing work often aims to flag useless checks with dedicated program analysis, while our approach generally circumvents this limitation. See Sec. 8 for comparisons with existing research.

Application Scenario. The focus and typical application scenario of SANRAZOR are to practically accelerate sanitization-enabled programs in production usage. When a production software cannot afford all the sanitizer checks, SANRAZOR can help effectively remove those checks that are least useful in terms of discovering unique problems. As will be shown in Sec. 6, SANRAZOR can reduce the overhead of ASan and UBSan significantly without primarily undermining vulnerability detectability. Moreover, SANRAZOR may be combined with complementary approaches to further reduce the overhead of sanitizer checks. For example, by combining SANRAZOR with ASAP, it is plausible to run these sanitizers in production (at 7% overhead) for their security benefits and vulnerability detectability. Thus, we believe users should generally incline to accept a low overhead (e.g., less than 10% when combining ASAP and SANRAZOR) for improved security and vulnerability detectability compared to running without sanitizer checks. In contrast, enabling full ASan can incur much higher cost (e.g., around 73.8%, as reported in Sec. 6) and is thus unrealistic in production. In practice, we would encourage users to explore combining SANRAZOR with other sanitization optimization tools [9, 37, 44] which share generally orthogonal focuses with SANRAZOR. We give further discussion and comparison with contemporary research works in Sec. 8.

4.1 Check Identification

We start by discussing how ASan and UBSan checks are identified. As aforementioned, each check can be represented as a comparison instruction followed by a control-flow transfer instruction in LLVM IR statements:

```
1 %o = icmp cond %a, %b
2 br i1 %o, label %bb1, label %bb2
```

where `%a` and `%b` are two LLVM IR identifiers, and `icmp` compares `%a` and `%b` w.r.t. the condition specified by `cond` (equal, greater than, etc.). A one-bit comparison output will be stored in `%o`, which is subsequently consumed by the control-flow instruction `br`. In case `%o` equals to one (i.e., the condition evaluates to "True"), the control flow will be transferred to the basic block pointed by `%bb1`; otherwise the basic block pointed by `%bb2` will be executed.

To distinguish sanitizer checks from user checks (i.e., branches in the source code), we search for calls to specific functions that are used by sanitizers. Specifically, a condition represents an ASan check, if a call to `_ASan_report` can be found in blocks pointed by label `%bb1` or `%bb2`. Similarly, a call to the `_UBSan_handle_XXX` function indicates the corresponding condition represents a UBSan check. Note that `XXX` denotes the name of an undefined behavior that this particular UBSan check detects. Overall, while ASan checks are designed to capture memory access errors, UBSan subsumes a much broader set of defects. The type of checked undefined behaviors can be seen from the handler name above, and indeed, the corresponding `icmp` statements can have different constant operands, characterizing the checked undefined behavior types.

4.2 Dynamic Check Pattern Capturing

SANRAZOR captures the dynamic patterns of checks by instrumenting the LLVM IR and inserting a counter statement before the `br` statement of identified checks (see the sample code in Sec. 4.1). We count how many times the control-flow statements are executed. We also record how many times the true and false branches are taken, by checking the one-bit operand of the control-flow statement. Sanitizer checks can be configured to abort the process or output an alert. To smoothly collect dynamic coverage patterns, we configure sanitizer checks to "alert users" instead of aborting the process. The collected coverage patterns will be used to identify redundant checks (cf. Sec. 4.4).

Workload Selection. Ideally, the more execution traces the selected workload can cover, the more comprehensive the dynamic patterns could become. SANRAZOR uses *default test cases* shipped with software to record dynamic patterns.

Our observation shows that the runtime overhead is primarily caused by sanitizer checks on *hot paths*. The shipped test cases typically suffice covering hot paths and derive nontrivial coverage patterns of most sanitizer checks. If one sanitizer check is never covered, it is *not* removed, since no dynamic pattern analysis is available. Sec. 7.4 presents further empirical evidences and discussions regarding workload selection.

4.3 Static Check Pattern Capturing

A user check or sanitizer check asserts certain program properties with a comparison statement (i.e., `icmp` in Sec. 4.1). For the static phase, we extract data-flow information from operands of each `icmp` statement. The extracted data flow facts constitute the static feature of a check.

To this end, SanRazor performs backward-dependency analysis to construct the data dependency graph for the branch condition operand. The analysis starts from the checked condition in the control-flow statement (the `br` instruction in Sec. 4.1; recall that `br` takes a LLVM identifier of one bit as its input). The checked condition register has a data dependency on two operands of the comparison statement. The constructed dependency tree will be used to constitute the static patterns of each check (see Sec. 4.3.1). The backward traversal will be stopped when we encounter terminal operands, including constants, `phi` [46] nodes, global variables, and function parameters. Also, when encountering function calls during the traversal (e.g., `malloc`), instead of performing heavyweight inter-procedural analysis, we take all function parameters as the dependency of the function return value.

4.3.1 Extracting Static Features with Three Schemes

After constructing the value dependency tree for the operand of the control-flow instruction `br`, the next step is to extract static features from the dependency tree. At this step, we design three schemes (*L0*, *L1*, and *L2*) by calibrating the extracted static features. Three schemes are designed as follows:

- *L0*, which gathers all the leaf nodes on the dependency tree into a set.
- *L1*, which canonicalizes the collected set of leaf nodes, by eliminating all constants from the set except constant operands from the comparison statement (`icmp` instruction) associated with each sanitizer or user check.
- *L2*, which canonicalizes the collected set of leaf nodes, by eliminating all constants from the set.

L0 collects all the leaf nodes into a set while *L1* and *L2* further canonicalize the constructed set by removing constant leaf nodes. In other words, we might treat checks for the following two pointer dereferences as “redundant”, although they check different program properties:

```
1  int a = *ptr;   // ASan check on ptr
2  int b = *(ptr + 4);   // ASan check on (ptr+4)
```

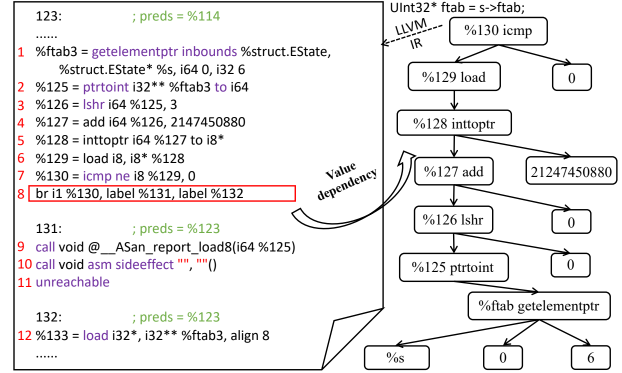

Figure 2: Tracing static data flow dependency.

With ASan enabled, the first and second ASan checks would take `ptr` and `ptr+4` as the inputs. Since both *L1* and *L2* schemes would eliminate constants (i.e., `4` for this case) from the leaf node set, these two checks are treated as redundant by *L1* and *L2*. Nonetheless, *L1* and *L2* schemes would unlikely miss discovering bugs derived from *pointer arithmetics*, in the sense that if an expression using pointer arithmetic (e.g., `ptr+4`) can provoke sanitizer check alerts, the pointer in the expression (i.e., `ptr`) is *presumably invalid* and provokes sanitizer check alerts as well. We present further discussion regarding security considerations in Sec. 4.3.2.

Overall, scheme *L1* and *L2* relax the notion of check “equivalence” by distilling pointer arithmetic expressions while still preserving rich information of the built dependency tree. Also, *L1* is designed to retain the constant operand of `icmp` statement associated with each check. As aforementioned, UBSan uses constant operands in the `icmp` statement to assert different program properties, keeping these specific constants can help to distinguish UBSan checks of different types. Sec. 6 further presents empirical results regarding each strategy; consistent with our intuition, evaluation results (Sec. 6.1) show that the relatively more aggressive scheme *L2* can help to identify more redundant checks and reduce runtime overhead. Moreover, Sec. 6.2 shows that even *L2* can still help to discover 33 (out of in total 38) CVEs from complex software. SanRazor provides all three schemes to extract static patterns, and we leave it to users to decide which one to use.

Fig. 2 illustrates feature extraction with an example. Once a path condition statement (line 8; the call statement on line 9 indicates this condition belongs to an ASan check) is identified, we trace the data dependency on the condition (`%130`) and construct a dependency tree. The dependency recovery forms a depth-first search, and we stop the search when encountering terminal nodes (e.g., the constants in Fig. 2). The recovered dependency tree will be used for comparison, following one of the schemes noted above.

4.3.2 Security Consideration

As previously mentioned, we deem that the *L1* and *L2* schemes would unlikely miss discovering bugs derived from

pointer arithmetics, given that if `ptr+4` is invalid, checking `ptr` is presumably sufficient to reveal the issue in practice. However, there are few corner cases, i.e., if `ptr` points to the end of an allocated memory, then `*ptr` is safe while `*(ptr+4)` corrupts. Similarly, `ptr` may point before the start of an allocated memory chunk: `*ptr` thus corrupts whereas `*(ptr+4)` is safe. As clarified in **Application Scenario** in Sec. 4, we focus on practically accelerating sanitization-enabled programs. Users concerned about "sophisticated attackers" can use *L0* or resort to full sanitization. When facing active attackers, another optimization opportunity is to first identify program attack interface, and then shave sanitizer checks out side those security sensitive code fragments. To do so, users can first employ information flow analysis techniques (e.g., taint analysis [31]). We leave it as future work to explore this direction.

4.3.3 Extension Using Static Analysis

As discussed in Sec. 3.2, computability theory suggests that it is difficult, and in general theoretically impossible, to rigorously establish the equivalence of two arbitrary nontrivial code fragments. Nevertheless, in practice, it is feasible to use static analysis to identify (likely) equivalent checks. In particular, we envision that symbolic techniques, e.g., (under-constrained) symbolic execution [27] and constraint solving, can be used to prove the *equivalence* of sanitizer checks.

We have observed a line of research seeking to perform code equivalence checking, by first collecting program input-output relations using symbolic execution [10, 19, 21]. Then, given symbolic constraints representing input-output relations of two code fragments, constraint solver can prove that these two code fragments are equivalent (suppose side effects are not considered). Moreover, constraint solvers can also be used to prove the inclusion of two symbolic constraints, i.e., deciding whether the satisfiability of one symbolic constraint will always induce the satisfiability of the other constraint. As a result, a potential extension of SANRAZOR is to decide whether check c_i validates a weaker property that can be inferred by a stronger property validated in another check c_j. If so, c_i could be redundant and removed. Overall, using such symbolic techniques could be the follow-up work of SANRAZOR to provide more principled guarantees; the tradeoff would be cost and scalability, given most symbolic execution-based code equivalent checking analyzes only basic blocks or execution traces [10, 20, 21].

In Sec. 7.2, we will show that the proposed technique can induce a number of false positive cases, i.e., treating distinct sanitizer checks as equivalent. However, most false positives discussed in Sec. 7.2 could be solved through intra-procedural static analysis, e.g., differentiating accesses to different fields in the same structure. We leave it as one future work to explore using field-sensitive point-to analysis (e.g., SVF [36, 38]) to alleviate false positives of SANRAZOR. Also, SANRAZOR currently omits to perform inter-procedural analysis, and as a result, sanitizer checks inside two procedures would be treated as different. This design decision may potentially lead to false negative cases (i.e., missing a pair of redundant checks). However, we find that in practice, false negative cases are primarily due to other reasons; see discussion in Sec. 7.3.

4.4 Sanitizer Check Reduction

For each pair of checks, SANRAZOR decides whether one check is redundant to the other, by comparing their static and dynamic patterns. In case two checks are identical w.r.t. *both* static and dynamic patterns, only one check will be retained.

Comparing Dynamic Coverage Patterns. Let the coverage pattern of sanitizer check sc_i be a tuple $\langle sb_i, stb_i, sfb_i \rangle$, where sb_i denotes the total coverage times of sc_i, stb_i and sfb_i represent that sc_i executes its true branch stb_i times and its false branch sfb_i times. Similarly, the dynamic pattern of a user-defined check uc_i can be denoted as a tuple $\langle ub_i, utb_i, ufb_i \rangle$, where ub_i denotes the total coverage times of uc_i, utb_i and ufb_i represent that uc_i executes its true branch utb_i times and its false branch ufb_i times. Then, for dynamic coverage patterns extracted from sanitizer check sc_i and user-defined check uc_i, if they satisfy one of the following conditions, we consider sc_i and uc_i having identical coverage patterns:

$$\begin{aligned}
&\text{(a)} \quad (sb_i = ub_i) \wedge \big((stb_i = utb_i) \vee (stb_i = ufb_i)\big) \\
&\text{(b)} \quad (sb_i = utb_i) \wedge \big((stb_i = sb_i) \vee (sfb_i = sb_i)\big) \\
&\text{(c)} \quad (sb_i = ufb_i) \wedge \big((stb_i = sb_i) \vee (sfb_i = sb_i)\big)
\end{aligned} \tag{1}$$

The first condition implies that two checks have the same dynamic pattern. This can happen when they reside on the same path and the sanitizer check sc_i's false branch has the same coverage times as the user check uc_i's true or false branch. As illustrated in Fig. 3(a), suppose sc_i and uc_i check the same program property, then the predicates of sc_i and uc_i shall be satisfied and failed for the same numbers of times. The latter two conditions are satisfied when sc_i is guarded by one branch of uc_i and one branch of sc_i is never executed. For instance, to understand the second condition (suppose sc_i and uc_i check the same property; see Fig. 3(b)), whenever uc_i is true, sc_i within its true branch (i.e., $sb_i = utb_i$) should always be evaluated to the same direction, as implied by $(stb_i = sb_i) \vee (sfb_i = sb_i)$.

Similarly, for two sanitizer checks sc_i and sc_j, if they satisfy the following condition, we assume that sc_i has identical dynamic patterns with sc_j (one case shown in Fig. 3(c)):

$$(sb_i = sb_j) \wedge \big((stb_i = stb_j) \vee (stb_i = sfb_j)\big) \tag{2}$$

As mentioned in Sec. 4.2, when collecting the dynamic coverage pattern, we configure sanitizer checks to "alert" users (not abort programs) and collect the coverage patterns for comparison. Depending on the implementation details, the program aborting/alerting routine could be found in either the true branch or the false branch of a sanitizer check. Hence, we

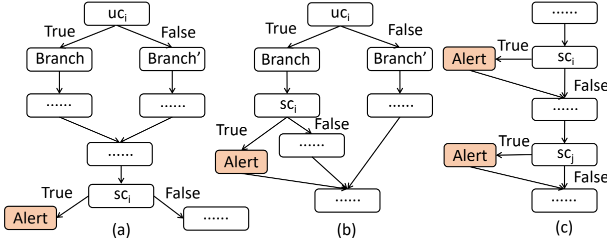

Figure 3: Coverage patterns. Sanitizer checks can be configured to abort programs or alert users. As noted in Sec. 4.2, we configure sanitizer checks to "alert" users (not abort programs) and collect the coverage patterns for comparison.

define a set of general conditions in Formulas 1 and 2, which shall take both cases (i.e., the "aborting" routine resides in true or false branches) into account.

We also note that we did not observe any alerts yielded by sanitizer checks in our experiments. In general, the alert branch of a sanitizer check is much less likely to execute, if at all, than the non-alert branch. The consequence is that a subset of the conditions in Formulas 1 and 2 are in fact used in our experiments. For instance, when the sanitizer check sc_i is within the true branch of uc_i, the second condition in Formula 1 must be satisfied. Nevertheless, in practice, this rarely affects the correctness of our redundancy judgement.

Comparing Static Dependency Patterns. As discussed in Sec. 4.3.1, we provide three schemes to extract static patterns (into sets) from dependency trees of operands in control transfer statements. Given two sets S_i and S_j formed by analyzing two checks c_i and c_j, c_i and c_j are considered to have identical static patterns, in case S_i and S_j are identical.

Removing Sanitizer Checks. SANRAZOR does *not* remove user-defined checks; we prune sanitizer checks in case they are redundant w.r.t. user or other sanitizer checks. Given a pair of likely redundant sanitizer checks c_i and c_j, we remove the check c_j if it was dominated by c_i. Users can also configure SANRAZOR to decide which one to remove. To remove a check, we set the condition of its control-flow statement (see Sec. 4.1) as `false` such that the branch for alerting/aborting will never be executed. This would let the dead-code-elimination of LLVM remove the redundant code.

Extension by Considering Dominating Cases. The aforementioned coverage pattern reasonably flags redundant checks and achieves high effectiveness of reducing overhead incurred by sanitizer checks. Nevertheless, we point that that Formula 2 only considers the equality cases; the dominating cases are not considered, which introduces false positives and the primary false negatives, as will be shown in Sec. 7.2 and Sec. 7.3. An improvement at this step is to maintain the potential dominating checks (denoted as $\mathcal{D}_i$) for sanitizer check c_i. Check c_i can be removed in case its dominating check $c_k \in \mathcal{D}_i$ manifests identical data dependency features with c_i. This extension primarily eliminates false negatives presented in Sec. 7.3.

5 Implementation

SANRAZOR [3] is written primarily in C++ with approximately 2,000 lines of code. We integrate SANRAZOR into the LLVM framework [17] by providing a wrapper of `clang`, namely `SanRazor-clang`. Users can replace `clang` in their building scripts with `SanRazor-clang`. `SanRazor-clang` inserts sanitizer checks to a C/C++ program, and then invokes our follow-up passes to reduce redundant checks. To use SANRAZOR, users need to prepare a reasonable amount of inputs. We note that standard test inputs would usually suffice removing a large amount of sanitizer checks; see our empirical study of workload selection in Sec. 7.4.

6 Evaluation

We give the cost evaluation of SANRAZOR in Sec. 6.1. We measure the reduction accuracy (in terms of vulnerability detectability) in Sec. 6.2 and compare it with ASAP in Sec. 6.3.

6.1 Cost Study

We start by measuring how well SANRAZOR can reduce the performance penalty of sanitizer checks. To this end, we leverage the industry-standard CPU-intensive benchmark suite, SPEC CPU2006, for the evaluation. SPEC CPU2006 contains 19 C/C++ programs. We are able to compile 11 SPEC benchmarks with the Clang compiler (version 9.0.0) and with ASan or UBSan enabled. These 11 test cases are `401.bzip2`, `429.mcf`, `445.gobmk`, `456.hmmer`, `458.sjeng`, `462.libquantum`, `433.milc`, `444.namd`, `470.lbm`, `482.sphinx3`, and `453.povray`. We encountered compatibility issues for the other benchmarks.[2]

Each SPEC benchmark is shipped with a training workload, a testing workload, and a reference workload. Following the convention, we use the training workload to profile these programs and obtain the dynamic patterns of sanitizer checks (see Sec. 4.2). After redundant sanitizer checks are removed by SANRAZOR, the reference workload is used to measure the performance of the optimized programs. We do not use test workload since it leads to much shorter execution time compared with the reference and training workload.

To evaluate the effectiveness of SANRAZOR, we measure the execution time reduction after eliminating redundant checks (referred to as M_0 metrics). We also count the number of removed sanitizer checks (referred to as M_1 metrics), and the execution cost (in terms of CPU cycles) saved by reducing sanitizer checks (referred to as M_2 metrics). M_0 is determined by measuring the execution CPU time. To calculate M_1, we record the total number of sanitizer checks inserted by the compiler and the number of sanitizer checks reduced by SANRAZOR. The calculation of M_2 is consistent with ASAP [40]

[2]Similar compatibility issues were also reported by ASAP [40]. We provide error messages in our artifact [3] for reference.

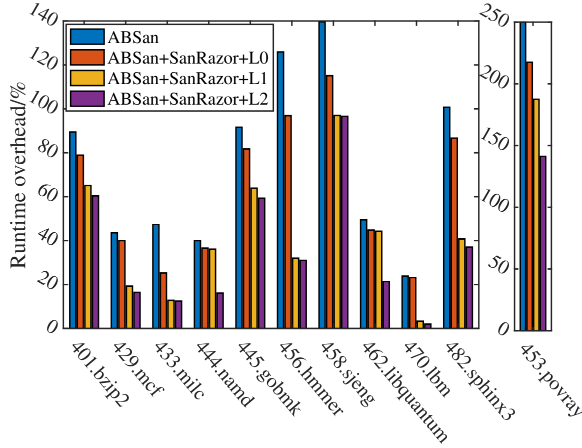

Figure 4: Comparison results w.r.t. M_0 metrics (execution time reduction) on ASan.

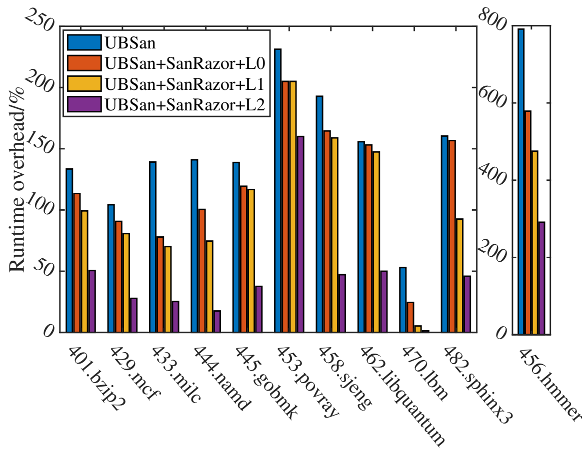

Figure 5: Comparison results w.r.t. M_0 metrics (execution time reduction) on UBSan.

(see comparison with ASAP in Sec. 6.3). In general, each sanitizer check performs a sequence of operations to assert a program property, and for each operation o_i (e.g., loading shadow memory), ASAP predefines a constant f_i denoting how many CPU cycles o_i takes. Suppose a sanitizer check is executed for c times and each execution requires in total F CPU cycles ($F = \sum_{i=1} f_i$), its execution cost is calculated as $c \times F$. We reuse f_i defined in ASAP to compute M_2.

Processing Time. All experiments are conducted on a workstation with an Intel i7-8700 CPU and 16GB memory. We use scripts provided by SPEC to profile programs and collect coverage patterns. It takes on average 323 CPU seconds to profile one SPEC program. The static dependency analysis phase of SANRAZOR takes on average 27 seconds per case.

Cost Evaluation Results

Fig. 4 reports the execution cost that is induced by ASan checks with, and without applying SANRAZOR. The geometric mean runtime overhead increase with full ASan enabled (the blue line) is 73.8%. After reducing redundant ASan checks, performance overhead is reduced by 12.1% (*L0*), 35.2% (*L1*), and 53.5% (*L2*), with the geometric mean remaining overhead being 62.0% (*L0*; the orange line), 35.8% (*L1*; the yellow line), and 28.0% (*L2*; the purple line). The reduced runtime overhead with the *L*2 scheme can be up to 91.7% (for `470.lbm`), and the smallest reduction (`445.sjeng`, which exhibits the highest overhead with ASan enabled) still reduces the runtime cost by 30.8%.

Fig. 5 also illustrates the performance overhead for UBSan. In general, the runtime overhead caused by UBSan (geometric mean 154.3%; see the blue line) is much higher than ASan. By reducing redundant checks, the performance overhead can be reduced by 13.7% (*L0*), 35.5% (*L1*), and 75.5% (*L2*), with the geometric mean remaining overhead being 124.4% (*L0*; the orange line), 94.7% (*L1*; the yellow line), and 36.6% (*L2*; the purple line). The reduced runtime overhead of *L2* strategies is up to 97.5% (`470.lbm`), and at least 62.1% (`401.bzip2`).

We measure how many sanitizer checks are reduced by SANRAZOR and the saved CPU cycles. Table 1 reports the portion of reduced checks w.r.t. the total number of checks (M_1). Similarly, it also reports the portion of saved CPU cycles during run time w.r.t. the total CPU cycles taken by sanitizer checks (M_2). SANRAZOR can eliminate on geometric mean up to 29.5% of the ASan checks for the SPEC programs (with *L2* applied), which leads to 41.0% less CPU cycle cost of ASan checks during run time. We also observed a similarly promising trend for the UBSan evaluation. As shown in Table 1, SANRAZOR can eliminate up to 39.3% of UBSan checks for the SPEC programs on geometric mean (with *L2* applied), corresponding to 77.0% less cost during run time.

Sanitizer checks contribute differently to the total execution cost (i.e., some checks are executed far more often than others). For instance, SANRAZOR (with *L2* enabled; see Table 1) eliminates 13.1% of ASan checks in `456.hmmer`. However, the M_2 metrics is reduced by up to 70.4%, indicating that the removed checks are on the program's *hot paths*. Our manual investigation confirms this intuition; more than 99.3% of the runtime overhead caused by ASan stems from one function `P7Viterbi` in module `fast_algorithms.c`, which contains intensive memory access checks within a loop. SANRAZOR successfully identifies many redundant sanitizer checks within this loop, inducing effective check reduction for `456.hmmer`.

We also find some reduced sanitizer checks on the cold path of the benchmarks. For instance, while 22.6% of the UBSan checks are removed from `462.libquantum`, these checks have low runtime coverage and therefore removing them does not significantly improve performance. An aligned trend can be seen from the performance evaluation of `462.libquantum` in Fig. 5. Our manual study indicates that for `462.libquantum`, sanitizer checks in `gates.c` (on the hot path) contribute more than 99.2% of the total runtime cost. However, these checks assert different undefined behaviors and cannot be flagged as "redundant" by SANRAZOR.

Table 1: Evaluation results w.r.t. M_1 (number of removed sanitizer checks) and M_2 (saved CPU cycles by reducing sanitizer checks). Note that "empty cells" for `imageworsener`, `zziplib`, `libzip`, `graphicsmagick`, `jasper`, `potrace`, and `mp3gsin` are *not* due to setup errors; they indicate those CVEs are not discovered by the corresponding ASan (or UBSan) checks.

Benchmark	ASan-M_1			ASan-M_2			UBSan-M_1			UBSan-M_2		
	L0	*L1*	*L2*	*L0*	*L1*	*L2*	*L0*	*L1*	*L2*	*L0*	*L1*	*L2*
`401.bzip2`	22.4%	54.4%	58.1%	4.3%	30.3%	34.2%	38.7%	54.8%	66.0%	27.3%	37.9%	68.1%
`429.mcf`	10.2%	53.0%	60.9%	3.0%	46.6%	60.1%	35.0%	51.8%	76.2%	37.8%	47.6%	86.0%
`445.gobmk`	5.2%	23.4%	26.6%	7.2%	33.7%	41.0%	12.6%	21.6%	51.3%	21.4%	23.3%	73.9%
`456.hmmer`	5.9%	11.7%	13.1%	14.4%	70.3%	70.4%	8.2%	11.0%	14.8%	49.2%	60.7%	78.3%
`458.sjeng`	5.9%	12.6%	13.4%	4.4%	34.4%	36.7%	12.1%	18.3%	51.0%	20.7%	25.2%	79.2%
`462.libquantum`	7.4%	16.3%	22.6%	0.8%	1.4%	2.4%	12.7%	15.6%	26.9%	0.8%	0.8%	58.8%
`433.milc`	23.5%	32.5%	33.5%	35.8%	80.9%	82.7%	27.6%	42.2%	54.6%	51.0%	60.6%	83.6%
`444.namd`	6.4%	18.9%	24.0%	10.2%	29.8%	57.7%	8.7%	16.0%	26.2%	40.4%	54.1%	84.8%
`470.lbm`	1.6%	68.5%	72.1%	0.0%	88.7%	92.5%	17.7%	48.2%	51.3%	46.0%	92.5%	97.6%
`482.sphinx3`	10.7%	27.1%	32.5%	2.5%	56.9%	58.3%	18.2%	23.7%	40.0%	11.9%	45.3%	67.2%
`453.povray`	7.2%	9.5%	21.2%	2.3%	12.1%	69.1%	11.1%	11.9%	22.6%	22.6%	24.0%	75.5%
`autotrace`	12.2%	27.6%	35.7%	22.4%	65.4%	73.1%	20.6%	25.2%	39.0%	48.6%	57.5%	78.3%
`imageworsener`	-	-	-	-	-	-	26.8%	37.1%	53.3%	17.8%	21.6%	64.0%
`lame`	9.5%	38.5%	40.8%	11.0%	57.5%	74.9%	23.3%	34.1%	47.5%	17.0%	46.6%	71.4%
`zziplib`	3.8%	20.4%	23.9%	12.9%	80.2%	90.3%	-	-	-	-	-	-
`libzip`	6.2%	19.9%	27.8%	1.0%	3.9%	44.9%	-	-	-	-	-	-
`graphicsmagick`	1.2%	4.5%	5.8%	20.1%	49.4%	63.3%	-	-	-	-	-	-
`tiff`	7.8%	21.7%	29.8%	0.2%	2.1%	2.6%	12.3%	15.8%	21.7%	7.6%	10.5%	65.6%
`jasper`	-	-	-	-	-	-	12.8%	17.3%	25.9%	19.6%	20.6%	69.6%
`potrace`	13.0%	31.2%	38.8%	5.4%	41.9%	48.7%	-	-	-	-	-	-
`mp3gsin`	11.6%	43.6%	46.0%	4.8%	74.8%	78.4%	-	-	-	-	-	-

6.2 Vulnerability Detectability Study

This section explores whether sanitizer checks marked as redundant are true positive w.r.t. Definition 1 given in Sec. 3.2. This study reflects the accuracy of SANRAZOR. We select a number of programs with CVE vulnerabilities from an actively-maintained CVE list [22, 23], which documents procedures to compile each program and reproduce its CVEs. We select programs based on whether it can be successfully compiled and whether their documented CVEs can be triggered by the shipped inputs, and whether those CVEs can be detected by ASan/UBSan.

Ten programs (with in total 38 CVEs) are used for this evaluation. These ten programs are not cherry-picked; when selecting these ten programs, we checked each program in the CVE program list from the beginning [22, 23] and skipped only those that could not be properly set up for our study. For the evaluation setup, we start by compiling the provided source code with ASan or UBSan enabled. We report that those 38 CVEs can all be triggered by at least one input provided by the CVE list [22, 23], and after enabling ASan or UBSan, all the CVE-triggering inputs can be captured by either ASan or UBSan (i.e., all CVEs can be discovered). We then use SANRAZOR to perform check reduction with three schemes (*L0*, *L1*, and *L2*) and check whether after pruning, the CVE-triggering inputs can still be captured. Table 2 reports the evaluation results in terms of which CVE vulnerabilities can still be discovered by the pruned checks.

The static analysis phase of SANRAZOR takes on average 17.5 CPU seconds to process one program. Programs in the CVE list are typically shipped with a small number of inputs, including both regular and bug-triggering inputs. At this step, we use regular inputs to generate dynamic coverage patterns and shave sanitizer checks. We then test if bugger-triggering inputs can still be captured by the remaining sanitizer checks. The execution of most programs takes negligible amount of time (on average 1.5 CPU seconds). Overall, their shipped inputs are used for asserting *functionality*, not for *benchmarking*. Regarding M_0 metrics, we report that SANRAZOR reduces geometric mean runtime overhead caused by ASan from 24.9% to 15.8–22.4% (depending on the different reduction schemes). Similarly, geometric mean overhead incurred by UBSan on these CVE programs is reduced from 7.0% to 1.5–5.0%. We also report that we observed significant hot-path vs. cold-path distinction of these CVE programs, given their inputs of relatively low comprehensiveness. Nevertheless, we note that in case a sanitizer check is not covered by a shipped regular input, we will not even have its dynamic coverage pattern, and thus will not remove it. This way, vulnerabilities relevant to this check can be protected.

We then evaluate these programs w.r.t. the M_1 and M_2 metrics. As shown in Table 1, there is no significant gap comparing the number of checks removed from the CVE and SPEC programs, e.g., 9.9% vs. 8.2% with ASan-M_1&*L0* and 19.1% vs. 19.2% with UBSan-M_2&*L0*. Therefore, Table 2 shows, even if approximate reduction is achieved, almost all CVEs can still be discovered. The rest of this section elaborates on each case. We discuss all *false positive* cases (i.e., missed CVEs due to incorrectly removed checks) exposed in Table 2 in Sec. 7.

`autotrace` is an open-source software written in C, transforming bitmap images into vector images. We reproduce

19 CVEs in six modules of `autotrace-0.31.1`. The UBSan checks avert nine CVEs, including eight signed integer overflows (CVE-2017-9161∼9163, CVE-2017-9183∼9187), and one left shift of negative value (CVE-2017-9188). The rest are heap buffer overflows detected by ASan checks (CVE-2017-9167∼9173, CVE-2017-9164∼9166). As reported in Table 2, all of these CVEs can still be detected, after eliminating redundant sanitizer checks with the *L0* and *L1* schemes. Nevertheless, *L2* generates two false positives for the UBSan cases (see Sec. 7.2).

`imageworsener` is a C/C++ library supporting scaling and processing images with multiple formats. We evaluate the performance of SANRAZOR with five CVEs found in `imageworsener-1.3.1`, including two divide-by-zero CVEs (CVE-2017-9201∼9202) in `imagew-cmd.c`, two null pointer dereferences (CVE-2017-9204∼9205) in `imagew-util.c`, and one out-of-bounds access (CVE-2017-9203) in `imagew-main.c`. As reported in Table 2, both *L0* and *L1* strategies in SANRAZOR can detect all these CVEs, while the *L2* strategy generates one false positive in CVE-2017-9203. As for the performance gain, Table 1 shows encouraging results by saving up to 64.0% w.r.t. M_2 metrics.

`lame` is a MP3 encoder written in C. We reproduce two CVEs in two C files of `lame-3.99.5`, where one is a division by zero vulnerability detected by UBSan in `get_audio.c` (CVE-2017-11720), and the other is a heap buffer overflow detected by ASan in `util.c` (CVE-2015-9101). Table 2 shows that both UBSan and ASan can still discover these two CVEs for all three settings. For the inserted ASan checks, Table 1 reports that up to 74.9% cost can be saved. Similar trends (up to 71.4%) can be observed for UBSan.

`zziplib` is a lightweight C library for extracting data from a zip file. Two CVEs (CVE-2017-5976∼5977) in `zziplib-0.13.62` are evaluated, and both of them are caused by heap buffer overflow in `memdisk.c`. Our evaluation shows that after check reduction with all three settings, both CVEs can still be discovered by ASan.

`libzip` is a C library for processing zip files. There is a use-after-free CVE (CVE-2017-12858) in `libzip-1.2.0`, whose triggering-inputs can be captured by ASan. As shown in Table 2, only the *L2* scheme mistakenly eliminates the corresponding ASan check and missed one CVE.

`graphicsmagick` is a tool for viewing and editing commonly-used file formats including PDF, PNG, and JPEG. We evaluate SANRAZOR on CVE-2017-12937, a heap use-after-free vulnerability in `sun.c`. Table 2 shows that this CVE can be detected by ASan checks for all three settings.

`libtiff` is a library for viewing and editing tiff images. Four CVEs, including two heap buffer overflows (CVE-2016-10270, CVE-2016-10271), one stack buffer overflow (CVE-2016-10095) and one division-by-zero (CVE-2017-7598), are used to evaluate SANRAZOR. Experimental results show that SANRAZOR can detect all CVEs in all three settings.

`jasper` is also a complex image processing tool (with over 40K LOC). We evaluate SANRAZOR on CVE-2017-5502, a left shift of a value less than zero that can be detected when UBSan checks are fully enabled. Our evaluation shows that after check reduction, this CVE can still be discovered by the remaining UBSan checks.

`potrace` is a commonly-used C tool for converting bitmaps into smooth and scalable images. We evaluate SANRAZOR on CVE-2017-7263, a heap buffer overflow in `bitmap.c` of `potrace-1.2`. Table 2 shows that ASan can still discover this CVE after redundant checks are removed in all three settings.

`mp3gain` is a C library for analyzing MP3 files. We reproduce five CVEs in `mp3gain-1.5.2`, including one null pointer dereference (CVE-2017-14406) in `interface.c`, one buffer overflow (CVE-2017-14407) in `gain_analysis.c`, and two buffer overflows (CVE-2017-14408∼14409) in `layer3.c`. As reported in Table 2, one false positive is found, where both the *L1* and *L2* schemes over-aggressively remove the sanitizer check for detecting CVE-2017-14406.

The evaluation has demonstrated the promising and practical accuracy of SANRAZOR: vulnerability detectability is unlikely impacted even if we reduce sanitizer cost. Also, readers may suspect that if during the profiling phase software is not "stressed enough", SANRAZOR will elide a small set of checks and, as expected, catch the respective CVEs. However, we again note that *all* sanitizer checks detecting the 38 CVEs are covered during the profiling phase. That is, our observation — at least 33 CVEs are discovered by the remaining checks — is not due to "under-stressed profiling", rather, the corresponding checks are not incorrectly deemed redundant.

6.3 Comparison Study

We compare SANRAZOR with the closely related work, ASAP [40]. ASAP reduces sanitizer checks with high cost in order to satisfy an overhead budget specified by users. In contrast to SANRAZOR, ASAP does not identify "likely identical checks". Given an overhead budget T, ASAP first estimates the performance cost that a sanitizer check c can incur. Sanitizer checks will then be ranked by cost and iteratively removed starting from the most expensive one until the estimated cost is lower than the budget T. We compare SANRAZOR with ASAP by running ASAP on our CVE cases with different overhead budgets, and reports whether the known CVEs can still be discovered (i.e., evaluation conducted in Sec. 6.2). Contrarily, we do *not* evaluate ASAP on the sanitizer overhead it can save. ASAP reduces sanitizer checks to meet a fixed cost budget; it is easy to see that comparing ASAP and SANRAZOR on this matter (i.e., SPEC evaluation in Sec. 6.1) is not reasonable.

In this evaluation, we start by using the default budget of ASAP, 5%, to measure the check reduction. As shown in Table 2, setting the overhead budget to 5% (i.e., Budget_0) prunes 23 out of 38 checks that can detect CVEs. Furthermore, to

Table 2: CVE case study. N denotes the number of CVEs. The "SANRAZOR" and "ASAP" columns report the number of remaining CVEs that can be discovered by sanitizer checks after reduction. ASAP allows to configure arbitrary overhead budget. We evaluate ASAP by using its default budget (Budget_0), and with the same overhead as SANRAZOR (i.e., Budget_1, Budget_2, Budget_3 correspond to *L0*, *L1*, *L2*, respectively). See Sec. 6.3 for the setup.

Software	CVE			SANRAZOR			ASAP			
	Type	Sanitizer	N	*L0*	*L1*	*L2*	Budget_0	Budget_1	Budget_2	Budget_3
`autotrace`	signed integer overflow	UBSan	8	8	8	6	6	8	8	8
	left shift of 128 by 24	UBSan	1	1	1	1	1	1	1	1
	heap buffer overflow	ASan	10	10	10	10	0	8	2	2
`imageworsener`	divide-by-zero	UBSan	2	2	2	2	2	2	2	2
	index out of bounds	UBSan	1	1	1	0	1	1	1	1
`lame`	divide-by-zero	UBSan	1	1	1	1	1	1	1	1
	heap buffer overflow	ASan	1	1	1	1	0	1	0	0
`zziplib`	heap buffer overflow	ASan	2	2	2	2	0	0	0	0
`libzip`	user after free	ASan	1	1	1	0	0	1	1	1
`graphicsmagick`	heap use after free	ASan	1	1	1	1	0	1	1	1
`libtiff`	heap buffer overflow	ASan	2	2	2	2	0	2	2	2
	stack buffer overflow	ASan	1	1	1	1	1	1	1	1
	divide-by-zero	UBSan	1	1	1	1	1	1	1	1
`jasper`	left shift of negative value	UBSan	1	1	1	1	1	1	1	1
`potrace`	heap buffer overflow	ASan	1	1	1	1	0	1	1	0
`mp3gain`	stack buffer overflow	ASan	2	2	2	2	0	2	0	0
	global buffer overflow	ASan	1	1	1	1	0	0	0	0
	null pointer dereference	ASan	1	1	0	0	1	1	1	1
In total			38	38	37	33	15	33	24	23

present a fair comparison with SANRAZOR, we iterate each tested program and put their CVEs into different types (the second column of Table 2). We then use the M_2 metrics of SANRAZOR in terms of each CVE type as three different overhead budgets of ASAP (i.e., budget_1, budget_2, budget_3). For instance, the "divide-by-zero" CVE of `imageworsener` can be captured by UBSan checks. After applying SANRAZOR (with *L0*, *L1*, and *L2* schemes) on `imageworsener` with full UBSan checks enabled, we report that the remaining M_2 overhead is 82.2%, 78.4%, and 36.0%, respectively. Then, ASAP is configured to take these three remaining M_2 overhead as its overhead budget and performs sanitizer check reduction. As shown in Table 2, two CVEs of `imageworsener` (the "divide-by-zero" row) can still be discovered for all three budgets. Overall, ASAP is configured to achieve the same amount of performance cost as SANRAZOR, and we record how many CVEs can still be discovered in this "apple-to-apple" setting.

We record the remaining M_2 overhead (geometric mean 89.2%) for each CVE type after using SANRAZOR with the *L0* scheme enabled. As shown in Table 2, five CVEs cannot be detected when assuming this budget for ASAP. The *L1* and *L2* schemes perform a relatively more tolerant reduction (73.5% and 45.7% geometric mean remaining M_2 overhead), and accordingly, ASAP removes 14 and 15 checks, respectively. We find that considerable critical CVEs are not discovered after using ASAP, since the corresponding checks are in the "hot paths" of test cases, incurring high cost, and therefore are removed. Overall, we interpret the comparison as encouraging, showing that ASAP neglects the important observation of sanitizer redundancy, causing it to fail discovering CVEs on hot paths. Contrarily, Table 2 shows that SANRAZOR can help discover more CVEs after reducing identical cost.

6.4 Combining SANRAZOR with ASAP

We also conduct a case study on `autotrace` to explore how we could achieve potential synergistic effects by combining SANRAZOR with ASAP and reduce the M_2 overhead for production usage. To this end, we explore whether, after applying ASAP, SANRAZOR can find further opportunities for eliminating redundancy that ASAP may have missed.

Specifically, we first set the overhead budget of ASAP to the reasonable, but arbitrary threshold of 30% and run it on `autotrace` with full ASan enabled to remove high-cost checks. ASAP aggressively reduces ASan checks; after reducing the M_2 overhead to 30%, six out of in total 10 CVEs are missed. We then leverage SANRAZOR to identify redundant checks. We report that when applying SANRAZOR with the *L*0 scheme, we observe that the M_2 overhead can be further reduced to 7.0%, *without missing any additional CVEs*. In contrast, using ASAP with this aggressive budget (7.0%), would reduce too many ASan checks and miss all 10 CVEs (cf. Table 2).

SANRAZOR Extension and Future Directions. Note that ASAP primarily focuses on shaving costly checks on the hot paths, which indicates promising potential of fine-tuning SANRAZOR's schemes to be more adaptive by taking cost into consideration. SANRAZOR can thus be extended to apply L0/L1 schemes to shave checks with low costs and L1/L2 schemes to shave checks with high costs. This should better balance performance and safety, rather than using the same scheme to all checks.

Our study in this section sheds light on the significant potential in combining SANRAZOR with previous works (e.g., [14,37,44]) and exploring their synergistic effects, since the strategies for which checks to remove are generally orthog-

Table 3: Quantitative analysis of the removed sanitizer checks.

Software	Sanitizer Type	#Reduced Checks	#Identical Checks	#Correlated Checks
401.bzip2	UBSan	11,406	6,562	4,844
autotrace	ASan	2,434	460	1,974

onal. Also, in addition to the combination strategy demonstrated above, we envision using other feasible schemes to combine SANRAZOR and ASAP. For instance, given a user-specified budget, we first use SANRAZOR to remove all the likely redundant checks, and if the budget is still not met, we use ASAP to remove further checks. We leave it as future work to explore other practical methods to combine SANRAZOR with existing sanitizer reduction tools.

This section demonstrates combining SANRAZOR with ASAP to reveal more debloating opportunities. In addition, we also expect that by using SANRAZOR to shave likely equivalent checks, sanitizer-guided security applications can be boosted. For instance, by shaving redundant checks, sanitizer-guided fuzz testing tool, ParmeSan [26], may have a higher throughput and likely find more bugs within a given time budget. We leave this as one future work to explore using SANRAZOR to boost ParmeSan.

7 Discussion

7.1 Characteristics of Removed Checks

This section further explains the characteristics of the checks that are removed. Given the observation that thousands of sanitizer checks are inserted into each program, we deem investigating all test cases infeasible. Rather, we manually checked SPEC program `401.bzip2` (with UBSan) and CVE program `autotrace` (with ASan) and analyzed check reduction patterns (Table 3). The two programs contain in total 24,132 checks (17,272 in `401.bzip2` and 6,860 in `autotrace`). SANRAZOR with the *L2* scheme enabled removes 66.0% sanitizer checks from `bzip2` and 35.7% checks from `autotrace`. We studied each removed check to identify two common patterns that we refer to as "identical checks" and "correlated checks". Below, we present typical cases for each category.

Checks that are identical with other checks. Sanitizer checks of this class have the same functionality as other checks. Consider the code snippet in `bzip2.c` as follows:

```
1 void BZ_blockSort(EState* s){
2  UInt32* ptr   = s→ptr;
3  UChar* block = s→block;
```

where two UBSan checks are inserted to check whether `s` is a null pointer. However, these two checks indeed assert the same property and are therefore identical with each other. Removing one of them can still ensure that the null pointer is detected. SANRAZOR will remove one of them since their coverage patterns are exactly the same and their control-flow statements (i.e., the `br` statement in LLVM IR) have condition operands of identical dependency trees.

Checks that are correlated with other checks. SANRAZOR removes this class of checks since they have the same dynamic and static patterns with other checks, indicating strong correlation with each other, as, for instance, for different pointer arithmetic expressions over the same pointer (as discussed in Sec. 4.3.1). Consider CVE-2017-9169 as an example:

```
1  *(temp++)= buffer[xpos * 3 + 2];  //line 353
2  *(temp++)= buffer[xpos * 3 + 1];  //line 354
3  *(temp++)= buffer[xpos * 3];  //line 355
```

which is a heap buffer overflow in line 353 of file `input_bmp.c` (`temp` in above code). When enabling ASan, three sanitizer checks (sc_1,sc_2,sc_3) are inserted for this case to check the shadow memory of pointer `temp` (in line 1-3 above). When using SANRAZOR (with *L1* or *L2* enabled) to analyze this case, all three checks exhibit identical dynamic and static patterns. Thus, two checks will be removed. Although the heap buffer overflow in CVE-2017-9169 roots in the invalid memory access of pointer `temp` in line 353, ASan can presumably detect the vulnerability when using any of the other two checks.

7.2 False Positive Analysis

SANRAZOR can also induce false positives (i.e., unique checks that are removed), since the captured dynamic patterns only provide statistical information of sanitizer checks and the static pattern sets used for the redundancy analysis could be optimistic as well. Below, we provide detailed analysis of all five false positive cases caused by the *L2* scheme of SANRAZOR (the false positive case of using *L1* scheme is also subsumed).

`CVE-2017-9203` is an index out of bounds vulnerability in `imagew-main.c` of `imageworsener-1.3.0` as follows:

```
1 int_ci = &ctx→intermed_ci[intermed_channel];
2 output_channel = int_ci→
      corresponding_output_channel;  // CVE
3 out_ci = &ctx→img2_ci[output_channel];
```

Specifically, `ctx` is an input argument of the enclosing function (line 2), which has a `struct iw_context*` type. When compiling this module with UBSan, three sanitizer checks will be inserted to detect index out of bounds, type check, and pointer overflow vulnerabilities on line 2. Recall as introduced in Sec. 4.1, these three UBSan checks can be differentiated by the constant operands of their associated `icmp` statement. Nevertheless, since all these checks take the memory address of `ctx` as its inputs, they have the same value dependency on `ctx` when using the *L2* scheme (recall *L2* eliminates all constants). Therefore, SANRAZOR will identify two of them as redundant sanitizer checks and remove them, causing UBSan to fail reporting the index out of bounds vulnerability in this CVE. However, if SANRAZOR is configured with *L1*, these checks can be kept since the constant parameter used to differentiate these three UBSan checks are preserved.

CVE-2017-12858 is a use-after-free vulnerability detected by ASan in zip_buffer.c of libzip-1.2.0. As shown below, variable buffer of zip_buffer_t* type attempts to access its element free_data in the if condition (line 5):

```
1  void _zip_buffer_free(zip_buffer_t *buffer){
2   if (buffer == NULL) return;
3   if(buffer→free_data){  // CVE
4    free(buffer→data);
```

When ASan is enabled, a check is inserted to assert whether the memory pointed by buffer has been freed before accessing its element free_data. SanRazor with the *L2* scheme enabled eliminates this check since it has the same dynamic coverage pattern with two user checks (two if conditions on line 2 and line 3), which also share the same data dependency pattern (since variable buffer is used for all three user and sanitizer checks). However, the check inserted by ASan performs shadow memory calculation, which indeed depends on different constant values with two user checks. Therefore, SanRazor with *L1* scheme can retain this check.

CVE-2017-9184 is a signed integer overflow in autotrace=0.31.1 reported by UBSan. It is derived from a heap memory allocation in input-bmp.c as follows:

```
1  XMALLOC(image, width * height
2          * 1 * sizeof(unsigned char));  // CVE
3  ypos = height - 1;
4  switch (...) {
5    case 1: {
6      while (ypos >= 0 && xpos <= width) { ...
```

XMALLOC allocates memory buffers on the heap by taking the second parameter as the buffer length, where a UBSan check is inserted in line 1 to detect the signed integer overflow when multiplying height with width. Moreover, we find a user check in the while loop condition (line 6), which shares the same value dependency with this critical sanitizer check (ypos derives from height and xpos is a constant). Therefore, SanRazor with *L2* enabled removes this check. Nevertheless, the inserted UBsan check and while condition assert different program properties, exposing a false positive. In contrast, this false positive can be avoided when using the *L1* scheme, since the user check also depends on constant value 0, exhibiting different value dependency patterns with the inserted sanitizer check by UBSan.

CVE-2017-9187 is a signed integer overflow vulnerability found from autotrace=0.31.1. Consider the following code snippet showing the CVE in input-bmp.c:

```
1   unsigned char *temp2, *temp3;
2   XMALLOC(image, width * height
3           * 3 * sizeof(unsigned char));  // CVE
4   temp3 = image;  //another UBSan check
```

When compiling this code snippet with UBSan enabled, a sanitizer check is inserted to check whether the second parameter of XMALLOC can incur an integer overflow. Also, another check is added to detect whether the pointer temp3 is null. Since temp3 points to image and the value of image is assigned by XMALLOC, the parameter of the second UBSan check depends on variable width and height (recall as mentioned in Sec. 4.3, for interprocedural analysis the function call output, image for this case, conservatively depends on all function parameters). Therefore, SanRazor with *L2* enabled will consider this sanitizer check to be redundant with the assertion, while the *L1* scheme would not, as these two checks can be differentiated by their constant parameters.

CVE-2017-14406 is a null pointer dereference found in interface.c of mp3gain-1.5.2. Consider the code below:

```
1  int sync_buffer(PMPSTR mp,int free_match) {
2   for (i=0; i<mp→bsize; i++) // CVE
3   { ... }
4   struct frame *fr = &mp→fr;
5   h = head_check(head,fr→lay);
```

Two ASan checks are used to check mp and fr when accessing their struct elements bsize (line 2) and lay (line 5), respectively. fr is initialized with mp->fr (line 4), which depends on the function parameter mp (line 1). That is, the two ASan checks have identical data dependencies w.r.t. the *L1* and *L2* schemes. SanRazor eliminates the first ASan check and becomes incapable of detecting the CVE vulnerability on line 2. However, SanRazor with *L0* enabled can differentiate these two checks, since they depend on different constant offsets when accessing the struct elements.

7.3 False Negative Analysis

SanRazor could also have false negatives (i.e., redundant checks are not removed). Take the following piece of code in 462.libquantum for example, where ASan inserts two checks to detect a buffer overflow in reg->node[i]. Let the inserted check in line 2 and line 3 as sc_1 and sc_2, respectively. Although sc_2 is redundant with sc_1 for this specific case, SanRazor does not recognize sc_2 as a redundant sanitizer check during our experiment, because the dynamic pattern of sc_2 differs from that of sc_1. Such cases are the primary cause for generating false negatives, according to our observations. Nevertheless, Sec. 4.4 has discussed that these false negative cases can be primarily eliminated by considering dominating relations in comparing dynamic coverage patterns.

```
1  for(i=0; i<reg→size; i++) {
2    if(reg→node[i].state & ...) {
3      if(reg→node[i].state & ...) {
```

7.4 Effects of Workload Selection

As mentioned in Sec. 4.2, SanRazor relies on dynamic coverage patterns to pinpoint potentially redundant checks. Therefore, in this section, we present study and discussion on the efficiency of check reduction w.r.t. the size of workload. To do so, we incrementally enlarge the workload for profiling bzip2 record both the number of reduced sanitizer checks

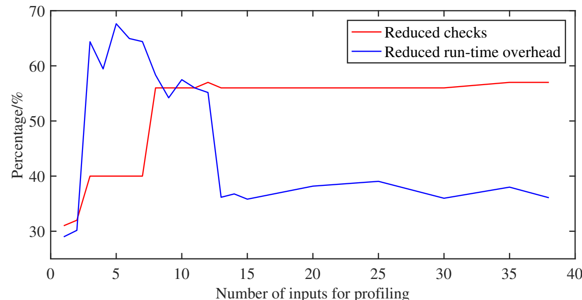

Figure 6: Effects of workload selection evaluation on ASan.

and the runtime overhead caused by ASan. For this study, we configure SANRAZOR with the *L2* scheme for check reduction. We download a `bzip2` testsuite with 38 different inputs from [2] and conduct experiments on `bzip2` (a single-file version from [1]).

As illustrated in Fig. 6, the percentage of reduced sanitizer checks will increase when more inputs are fed to the test case. However, when the number of test inputs are more than eight, the number of reduced checks reach the saturation point at 58.0%. Similarly, the percentage of reduced runtime cost can also become stable when the number of inputs is larger than 12. Also, notice that the blue line will first increase (when the input is less than five), and then decrease until reaching the saturation point (when input is 13). The reason is that with insufficient amount of inputs in the workload, irrelevant checks may exhibit identical dynamic coverage patterns and be treated as "redundant." In other words, SANRAZOR may report false positives when the available inputs are insufficient, and aggressively flag too many "redundant" checks (Sec. 7 discusses false positives and false negatives of this research). In general, when adopting SANRAZOR in real-world scenarios, sufficient inputs are needed for achieving good reduction results and reducing false alarms, but they may not need to be too many.

8 Related Work

Static Check Reduction. Existing research has proposed heavyweight program analyses to elide redundant bounds check by inferring the value ranges of certain variables. For example, some approaches deem checks unnecessary if the value range of an index is below the size of its accessed array [6, 11, 12, 15, 25, 37, 39, 44, 45]. SIMBER [7, 45] uses statistical inference to identify redundant bounds checks from past executions. RedCard [9] flags unnecessary race condition checks by scoping specific "release-free" code region where it is proved that only one race check is needed for each region. BigFoot [28] coalesces race checks on arrays and C structs. Overall, the extensive existing work on this topic focuses on specific types of checks, and cannot be easily generalized to other checks. For instance, [9] identifies a "release-free" region by checking if no lock release synchronization operations (e.g., wait, fork) can be found in that region. Hence, race checks only need to be done once within each region. Scoping such a "safe region" could be very difficult for other checks: to decide such a safe region for ASan, we anticipate to perform expensive alias analysis for every pointer within that region to confirm a checked pointer is never modified. In contrast, SANRAZOR analyze the equivalence of checks in a general and practical way to eliminate duplications.

The most closely related work is ASAP [40], which, like SANRAZOR, is unsound but general. ASAP is designed based on the observation that a few "hot" sanitizer checks account for most of the overhead and that most CVEs are located in the "cold" parts of a program. ASAP removes sanitizer checks with high runtime overhead until the overall overhead meets a user-provided cost budget. Different from SANRAZOR, ASAP does not consider check redundancy and is prone to removing critical checks on a program's hot paths as we have shown.

Run-time Check Reduction. Safe Sulong [30] is a sanitizer that relies on the dynamic compiler of the Java Virtual Machine to reduce checks. Java compilers are capable of eliding certain unneeded checks [42]. However, Safe Sulong can be overly conservative since it eliminates only those checks that are identified as redundant by the compiler.

Some approaches reduce sanitizer checks by runtime partitioning. Kurmus et al. split the kernel into an unprotected and a protected partition to reduce overhead caused by kernel hardening [16]. Bunshin [43] distributes sanitizer checks into different program variants and executes them in parallel to reduce the overall overhead. PartiSan [18] is a runtime sanitizer partitioning tool using control-flow diversity to improve sanitizer efficiency. Varan [13] is a multi-version monitor that uses selective binary rewriting to execute multiple versions of a system (e.g., instrumented by multiple sanitizers). However, these approaches reduce sanitization cost with parallel execution, rather than analyzing redundancy offline.

9 Conclusion

We have presented SANRAZOR, a novel, practical tool for sanitizer check reduction. SANRAZOR identifies redundant checks by analyzing their dynamic coverage patterns and static data dependency patterns. Evaluation on CPU benchmarks and programs with CVEs shows that SANRAZOR can effectively lower the overhead caused by ASan and UBSan, while still retaining high vulnerability detection capability.

Acknowledgments

We thank anonymous reviewers and our shepherd, Shan Lu, for their valuable feedback. We also thank Fuqiang Fan, who proofread an early version of the paper and pointed out an error in Definition 1.

References

[1] Large single compilation-unit C programs. https://people.csail.mit.edu/smcc/projects/single-file-programs/, 2006.

[2] Bzip2 testsuite. https://sourceware.org/git/?p=bzip2-tests.git, 2019.

[3] SanRazor. https://github.com/SanRazor-repo/SanRazor, 2021.

[4] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy*, pages 263–277. IEEE, 2008.

[5] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997.

[6] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 321–333, 2000.

[7] Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. CHOP: Bypassing runtime bounds checking through convex hull optimization. *Computers & Security*, 90:101708, 2020.

[8] LLVM Developers. Undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2017.

[9] Cormac Flanagan and Stephen N Freund. Redcard: Redundant check elimination for dynamic race detectors. In *European Conference on Object-Oriented Programming*, pages 255–280. Springer, 2013.

[10] Debin Gao, Michael K. Reiter, and Dawn Song. BinHunt: Automatically finding semantic differences in binary programs. ICICS, 2008.

[11] Rigel Gjomemo, Phu H Phung, Edmund Ballou, Kedar S Namjoshi, VN Venkatakrishnan, and Lenore Zuck. Leveraging static analysis tools for improving usability of memory error sanitization compilers. In *International Conference on Software Quality, Reliability and Security (QRS)*, pages 323–334, 2016.

[12] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering*, (3):243–250, 1977.

[13] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient N-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 339–353, 2015.

[14] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient sanitizer metadata design for fuzzing. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 249–263, 2020.

[15] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 270–278, 1995.

[16] Anil Kurmus and Robby Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *ACM Conference on Computer & Communications Security (CCS)*, 2014.

[17] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–, 2004.

[18] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. PartiSan: fast and flexible sanitization via run-time partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 403–422, 2018.

[19] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. Measuring code behavioral similarity for programming and software engineering education. In *International Conference on Software Engineering Companion (ICSE-C)*, pages 501–510, 2016.

[20] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. Automatic grading of programming assignments: an approach based on formal semantics. In *International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 126–137, 2019.

[21] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE*, 2014.

[22] Dongliang Mu. CVE list. https://github.com/VulnReproduction/VulnReproduction.github.io, 2019.

[23] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, 2018.

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 245–258, 2009.

[25] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 333–344, 1998.

[26] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.

[27] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[28] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. Bigfoot: Static check placement for dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 141–156, 2017.

[29] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[30] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and thanks for all the bugs: Finding errors in C programs by abstracting from the native execution model. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 377–391, 2018.

[31] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, 2010.

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, 2012.

[33] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. 2019.

[34] Cloyce D Spradling. SPEC CPU2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 35(1):130–134, 2007.

[35] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, 2015.

[36] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *International Conference on Compiler Construction (CC)*, pages 265–266, 2016.

[37] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016.

[38] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE TSE*, 40(2):107–122, 2014.

[39] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, pages 132–143, 1977.

[40] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *IEEE Symposium on Security and Privacy*, pages 866–879, 2015.

[41] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 260–275, 2013.

[42] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the Java HotSpot client compiler. In *International Symposium on Principles and Practice of Programming in Java (PPPJ)*, pages 125–133, 2007.

[43] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 271–283, 2017.

[44] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 70–82, 2000.

[45] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 413–426, 2017.

[46] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. pages 175–186, 2013.