Evaluating the Effectiveness of Memory Safety Sanitizers

Emanuel Q. Vintila
Technical University of Munich
emanuel.vintila@tum.de

Philipp Zieris
Fraunhofer AISEC
philipp.zieris@aisec.fraunhofer.de

Julian Horsch
Fraunhofer AISEC
julian.horsch@aisec.fraunhofer.de

***Abstract*—C and C++ are programming languages designed for developing high-performance applications, such as web browsers and operating systems. This performance is partly achieved by sacrificing memory safety, which introduces the risk of memory bugs—the root cause of many of today's most severe vulnerabilities. Numerous solutions have been proposed to detect and prevent memory bugs, with the most effective employing dynamic program analysis to sanitize memory accesses. These memory safety sanitizers vary greatly in their capabilities, covering different memory regions and detecting different subsets of memory bugs. While conceptual classifications of these sanitizers exist, practical and quantitative evaluations have primarily focused on performance rather than their actual bug-finding capabilities.**

To bridge this gap, we present MSET, a tool for evaluating memory safety sanitizers, along with an extensive functional evaluation of the most powerful and widely used memory safety sanitizers. We systematically deconstruct memory safety bugs into distinct properties, such as the memory region, the method of memory corruption, and the type of access to the target buffer. Using this systematization, our tool generates test cases that combine small and unique code templates, covering all typical memory bugs, including various forms of buffer overflows, underflows, and use-after-frees. Our functional evaluation highlights the differences between the conceptual detection potential of sanitization techniques and the bug-finding capabilities of sanitizers with similar objectives. Furthermore, it reveals that multiple sanitizers fail to achieve their conceptual potential due to incomplete or faulty implementations. Our tool is available as open source software, enabling researchers and practitioners to test their sanitizers and uncover lost potential, conceptual shortcomings, and implementation errors.

1. Introduction

C and C++ are system programming languages known for their speed and memory efficiency. However, these advantages come at a cost, as C and C++ are prone to memory bugs, which can lead to severe vulnerabilities such as data corruption, information leaks, or control-flow hijacking [1]. Studies conducted by Microsoft [2] and Google [3] revealed that memory bugs remain prevalent and highly exploitable in modern software. The White House has issued a statement [4] urging the industry to acknowledge the significant risks associated with vulnerabilities arising from memory bugs.

Memory bugs can be classified into two categories: *temporal bugs*, which occur when pointers reference objects that have been deallocated (e.g., use-after-free accesses), and *spatial bugs*, which involve accessing objects outside their allocated memory (e.g., buffer overflows). To detect memory bugs, numerous solutions have been proposed by both the research community and the industry. These solutions can be divided into two categories: those aimed at finding memory bugs during testing and those focused on mitigating them in actively used programs. For testing, static code analysis tools examine the code without executing it, identifying potential execution paths where memory bugs may arise [5], [6]. In contrast, dynamic program analysis tools monitor and control the program during runtime, considering the runtime context to identify memory bugs [7]. Consequently, they can be utilized for both testing and productive use. In this paper, we focus on dynamic solutions, specifically *memory (safety) sanitizers*. These sanitizers can be directly compiled into C and C++ programs or loaded as shared libraries.

The adoption of memory safety sanitizers—particularly for software testing—has been steadily growing in recent years. Sanitizers such as ASan [18] have become integral parts of major compilers and are now easier to use than ever. However, a pertinent question arises: Are deployed sanitizers reliably detecting memory bugs in software projects? Unfortunately, the community has yet to establish a widely accepted and user-friendly metric to assess the quality of memory sanitizers. As a result, authors have employed various methods to evaluate the effectiveness of their sanitizers and only conceptual comparisons of sanitizers have been possible thus far [7]. For our work, we consider 45 sanitizers, using the comprehensive list provided by Song et al. [7] and adding 18 prominent or innovative sanitizers from recent years. We focus exclusively on memory safety sanitizers, i.e., those that can detect or prevent spatial or temporal memory bugs. Sanitizers targeting other types of bugs, such as uninitialized reads or race conditions, are not considered. Table 1 lists the sanitizers along with a summary of the functional evaluations provided by their respective authors, sorted by spatial bug detection techniques.

Among the 45 sanitizers considered, only 22 were published with evaluations using purpose-built test suites that theoretically yield consistent and comparable results. How-

© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/SP61157.2025.00088

TABLE 1. Overview of memory safety sanitizers

Sanitizer	Year Published	Spatial technique	Temporal technique	Available online	Tested /w MSET	Number of test cases used by authors	Detection rate reported by authors	Real-world bugs
Electric Fence [8]	’93	*G*	*I*	✓	✓			
PageHeap [9]	’00	*G*	*I*					
D&A Dangling [10]	’06	*G*	*O*					
Scudo [11]	’16	*G*	*I*	✓	✓			
FreeGuard [12]	’17	*G*	*I*	✓	✓			✓
Oscar [13]	’17	*G*	*O*					
Purify [14]	’92	*R*	*I*					
Memcheck [15]	’05	*R*	*I*	✓	✓			✓*
Dr. Memory [16]	’11	*R*	*I*	✓	✓			✓*
LBC [17]	’12	*R*		✓		869[B,R,S]	90.3%	✓
ASan [18]	’12	*R*	*I*	✓	✓			✓*
QASan [19]	’20	*R*	*I*	✓	✓	12515[J]	77.5%	✓
RetroWrite [20]	’20	*R*	*I*	✓		5871[J]	55.45%	
ASan-- [21]	’22	*R*	*I*	✓	✓	5644[J]	100%	✓
J&K [22]	’97	*O*						✓*
CRED [23]	’04	*O*				20[W]	100%	✓
D&A Bounds [24]	’06	*O*				14[Z]	100%	
BBC [25]	’09	*O*				18[W]	94.4%	
PAriCheck [26]	’10	*O*						
LowFat [27], [28], [29]	’17	*O*		✓	✓	87[C,R,W]	96.5%	✓
CUP [30]	’18	*O*	*L*			4038[J]	100%	
HWASAN [31]	’18	*O*	*L*	✓	✓			
RedFat [32]	’21	*O*	*I*	✓	✓	484[C,J]	100%	
PACMem [33]	’22	*O*	*L*			11531[J]	99.4%	✓
CryptSan [34]	’23	*O*	*L*	✓		5364[J]	98.7%	✓
MTSan [35]	’23	*O*	*L*			11378[J]	75.83%	✓
CAMP [36]	’24	*O*	*T*	✓		5372[C,J]	100%	✓
RTCC [37]	’92	*P*						✓*
P&F [38]	’97	*P*	*L*					✓*
MSCC [39]	’04	*P*	*L*					✓*
SoftBound [40]	’09	*P*		✓	✓	33[B,W]	100%	✓
IPC [41]	’12	*P*	*T*					
Delta Pointers [42]	’18	*P*		✓	✓	8[C]	87.5%	
EffectiveSan [43]	’18	*P*	*I*	✓	✓			✓*
SoftBound+CETS [44]	’24	*P*	*L*	✓	✓	5438[J]	100%	
FFmalloc [45]	’21		*O*	✓				✓
DangZero [46]	’22		*O*	✓		n/a[J]	100%	✓
PUMM [47]	’23		*O*	✓		33[C,U]	100%	✓
Undangle [48]	’12		*T*					✓*
DangNull [49]	’15		*T*					✓
FreeSentry [50]	’15		*T*	✓		5[C]	100%	✓*
DangSan [51]	’17		*T*					✓
MarkUs [52]	’20		*T*	✓				
CETS [53]	’10		*L*	✓	✓	30[S]	100%	
PTAuth [54]	’22		*L*	✓		150[J]	100%	✓

Spatial techniques: *G* Guard pages; *O* Per-object bounds tracking; *P* Per-pointer bounds tracking; *R* Red-zones

Temporal techniques: *I* Deallocated memory invalidation; *L* Lock-and-key; *O* One-time allocators; *T* Dangling pointer tagging

Test suites: [B]BugBench [55]; [C]Derived from CVE database [56]; [J]Juliet Test Suite [57]; [R]RIPE [58]; [S]Predecessor of Juliet [57]; [U]UAFBench [59]; [W]Wilander et al. [60]; [Z]Zitser et al. [61]; *Authors have also detected previously unknown bugs

ever, in practice, we found that no particular test suite has been used for more than 12 sanitizers. This can be attributed to the lack of general acceptance of these test suites and their unsuitability for the functional testing of memory safety sanitizers. The test suites used often focus on evaluating Control-Flow Integrity (CFI) [62], [63] rather than memory safety (RIPE [58], Wilander et al. [60]), are designed for evaluating static code analysis tools (Juliet Test Suite [57]), or lack specificity for precise memory safety evaluation (BugBench [55], UAFBench [59], Zitser et al. [61]), as discussed in Sections 3 and 7. This is supported by the high reported detection rates, often reaching 100%, for many sanitizers evaluated with these suites. Furthermore, when using diverse collections of test cases such as the Juliet Test Suite, which covers various categories from the Common Weakness Enumeration (CWE) [64], most of which are unrelated to memory safety, authors tend to select—and sometimes are forced to, due to incompatibilities—only a subset of relevant test cases. This high variability in test suites leads to vastly inconsistent results, making it virtually impossible to compare the capabilities of these sanitizers. In addition to test suites, 27 sanitizers have been evaluated on real-world bugs. While these results are valuable, authors rarely provide detailed information about the types of memory bugs that escaped detection, further complicating comparisons among sanitizers.

In this paper, we aim to bridge this gap by introducing a novel Memory Sanitizer Evaluation Tool (MSET)[1] that enables the measurement of sanitizer effectiveness through functional testing. Additionally, we provide a comprehensive comparison of existing memory sanitizers. From the 45 sanitizers listed in Table 1, we evaluate 16 for which suitable implementations are available online (see Table 1, and Table 2 for the versions used) and compare their actual bug finding capabilities with their conceptual capabilities, revealing significant differences in many cases. The remaining 10 sanitizers with implementations available online could not be evaluated, either because they were non-functional, required specific hardware, or were designed as mitigations that cannot detect bugs (refer to Section A of the Appendix for details). Our evaluation tool generates small C/C++ test cases, each containing a memory bug, as a combination of code templates from multiple orthogonal bug dimensions. The tool assesses each sanitizer based on its ability to detect these memory bugs. The simplicity of the generated code allows for the evaluation of both fully developed sanitizers and those that support only a basic set of C/C++ features (e.g., lacking support for multi-threading or vectorization). In total, MSET generates 232 test cases in 820 variants and can evaluate a variety of sanitizers, including compiler-based universal solutions such as ASan [18], hardened heap allocators such as Scudo [11], and hardware-assisted sanitizers such as HWASAN [31].

Our evaluation reveals that none of the tested sanitizers provide complete memory safety in practice. ASan [18], likely the most popular sanitizer, achieves average detection

1. Source code: www.github.com/Fraunhofer-AISEC/MSET

rates of 54.25% for spatial safety and 83.3% for temporal safety. SoftBound [40] and EffectiveSan [43], which are conceptually the most complete sanitizers, achieve average detection rates of 78.3% and 94.1% for spatial safety, respectively. Our results confirm the necessity of functional testing in sanitizer development, as relying solely on conceptual evaluations can overlook practical implementation challenges, resulting in less effective sanitizers. Furthermore, our work quantifies the theoretical capabilities of different sanitizer techniques, enabling us to rigorously compare not only the sanitizers themselves but also their underlying concepts.

The paper is structured as follows: Section 2 presents an update to the sanitizer classification by Song et al. [7] and explains the conceptual effectiveness of the different techniques used by sanitizers to detect memory bugs. In Section 3, we discuss our taxonomy of memory bugs and describe our evaluation methodology. Section 4 introduces MSET, the implementation of our evaluation methodology. The evaluation of each of the 16 sanitizers is detailed in Section 5. In Section 6, we discuss the limitations of MSET. Finally, in Section 7, we compare our approach to other functional evaluation methods before concluding our paper in Section 8.

2. Memory Safety Sanitizers

Dynamic program analysis for sanitizing memory accesses involves instrumenting programs to monitor their state at runtime and detect illegal behavior immediately. Song et al. [7] provide a comprehensive taxonomy and comparison of sanitizers, to which we provide an update in this section. Later in this paper, we utilize this updated taxonomy to compare the conceptual bug finding potentials of sanitizers with their actual bug finding capabilities. As our evaluation focuses specifically on spatial and temporal memory safety sanitizers, we exclude other types of bug finding techniques from our taxonomy update, such as type error detection or uninitialized data use detection. An overview of sanitizers and their bug finding techniques, including those categorized by Song et al. and noteworthy new publications from recent years, is presented in Table 1.

2.1. Spatial Memory Bug Finding Techniques

Spatial memory safety sanitizers detect dereferences of pointers that access memory outside the bounds of the object from whose base address they are derived (their *intended referent*). They can be categorized as *location-based* or *identity-based*: while location-based sanitizers simply track inaccessible memory outside the bounds of objects, identity-based sanitizers explicitly track accessible memory, maintaining a relationship between a pointer and its intended referent.

2.1.1. Location-based Sanitizers. Location-based sanitizers track inaccessible memory regions using *guard pages* or *red-zones* and report memory violations in case of an access to those regions. Guard pages [8], [9], [10], [11], [12], [13] safeguard objects by placing inaccessible memory pages before and/or after each object. Two noteworthy sanitizers not included in Song et al.'s taxonomy are Scudo [11] and FreeGuard [12]. To enhance performance, both sanitizers avoid placing guard pages thoroughly between every object. Instead, FreeGuard opts for a random distribution of guard pages throughout the heap, while Scudo, a hardened allocator rather than a classical sanitizer, places guard pages solely between large objects and dedicated memory regions containing only same-sized small objects.

Other approaches use red-zones [14], [15], [16], [17], [18], [19], [20], [21], which are small chunks of memory marked inaccessible and placed between objects. Sanitizers employing this technique detect memory bugs by maintaining a validity state for each memory byte in a *shadow memory* and evaluating this state to determine the legality of memory accesses. The detection capability of red-zones is generally superior to that of guard pages, as red-zones can be byte-precise and do not necessitate object padding. Due to the maturity of ASan [18] and its wide-spread adoption in real-world applications (especially fuzzing), no notable new sanitizers using this technique have been published in recent years. However, ASan has seen various adaptations and updates, although none of which alter its fundamental bug finding technique. It has been adapted for QEMU as QASan [19], implemented as the binary rewriter RetroWrite [20], and optimized for performance as ASan-- [21].

While location-based sanitizers are generally faster than identity-based sanitizers, they often lack precision; they may not detect spatial memory bugs such as indexing errors that bypass red-zones and guard pages. Additionally, to avoid disrupting the memory layout of programs, location-based sanitizers typically do not add red-zones between members of the same compound object, making *intra-object* memory bugs undetectable.

2.1.2. Identity-based Sanitizers. An alternative approach to finding spatial memory bugs involves tracking object identities by maintaining the bounds of each object in shadow memory or custom metadata structures. This enables sanitizers to conceptually detect all types of spatial memory errors, including intra-object bugs.

Object bounds can be tracked per object or per pointer. *Per-object bounds tracking* sanitizers [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [35], [36] employ various techniques to maintain object metadata and link pointers to the metadata of their intended referent. Since Song et al., *low-fat pointers* [65] and *pointer tagging* have prevailed, leading to new sanitizer development. Sanitizers using low-fat pointers implicitly link pointers to their intended referents by dividing the program's memory into equally sized chunks, each supporting a fixed allocation size. With a low-fat pointer, the intended referent's base address and size can be trivially inferred from the chunk's base address and its fixed allocation size. RedFat [32], the logical successor to LowFat [27], [28], [29], utilizes low-fat pointers to look up object metadata (validity state and allocation

size) embedded in red-zones placed immediately before each object, overcoming LowFat's inability to detect temporal memory bugs (see Section 2.2.1) and its requirement for object padding (and thus its inability to detect overflows into the padding). CAMP [36] reverts to using the implicitly encoded base and size information from its low-fat pointers, but instead of monitoring pointer dereferences, it monitors pointer arithmetic to validate the bounds of the derived pointer using the object's base address and size inferred from the original pointer.

Pointer tagging sanitizers repurpose the unused most significant bits of a pointer to store a link (called *tag*) to the intended referent's metadata. First, we correct a minor error in Song et al.'s taxonomy, which did not categorize CUP [30] as a per-object bounds tracker: CUP implements a special form of pointer tagging, replacing the most significant 32 bits of a pointer with an index into a disjoint *object* metadata table, while only storing the pointer's offset from the intended referent's base address in the remaining 32 bits. Similarly, PACMem [33] maintains a disjoint metadata table and utilizes the ARM Pointer Authentication (PA) feature to create a signature of the object's metadata, embedding it into the object's pointer as a tag and using it as an index to the table. HWASAN [31] uses per-object identifiers as tags and tracks the bounds of objects by storing the identifier of an object in shadow memory, where the identifier occupies space equivalent to the object's size. Building on this, CryptSan [34] uses ARM PA to create cryptographic tags from the object identifiers, enhancing its security properties and improving performance. MTSan [35], a binary rewriter designed for fuzzing, also uses object identifiers similar to HWASAN but adjusts its identifiers with each fuzzing run as it gradually learns the bounds of objects.

Per-object bounds tracking sanitizers do not detect intra-object overflows, as they cannot differentiate between members of a compound object. To address this, *per-pointer bounds tracking* sanitizers [37], [38], [39], [40], [44], [41], [42], [43] explicitly attach bounds metadata to each pointer and propagate it during pointer arithmetic or assignments. By narrowing the bounds of sub-object pointers, these sanitizers conceptually offer complete spatial memory safety. Since the publication of SoftBound [40], the best-known and conceptually most mature representative of this technique, per-pointer bounds tracking has seen little innovation. In 2024, SoftBound received a revision [44]: it was ported to a newer compiler version and several implementation flaws, such as missing intra-object bug detection, were addressed. A noteworthy sanitizer since Song et al. is Delta Pointers [42], which relies solely on pointer tagging. It divides pointers into an overflow bit, a 31-bit tag, and a 32-bit address, where the tag encodes the negative distance from the end of the intended referent to the current offset of the pointer. Pointer arithmetic on the address is also performed on the tag, making overflows detectable as the most significant bit, the overflow bit, is set when the distance becomes positive.

EffectiveSan [43], categorized by Song et al. as a type safety sanitizer, also indirectly provides per-pointer bounds tracking. It uses the low-fat pointer scheme to bind comprehensive type metadata, including metadata for sub-types of compound types, to pointers. EffectiveSan uses a pointer's offset into its referenced object to perform a type check on the sub-object the pointer refers to. Upon successfully passing this type check, it calculates the bounds of the referenced sub-object from the stored sub-type metadata to check the spatial validity of the intra-object access.

2.2. Temporal Memory Bug Finding Techniques

Temporal memory safety sanitizers detect dereferences of pointers whose intended referents have already been deallocated, as well as attempts to deallocate objects through invalid pointers. Song et al. identify three techniques—*reuse delay*, *dangling pointer tagging*, and *lock-and-key*—that these sanitizers utilize to achieve their objectives. In our update to the taxonomy, we categorize sanitizers that delay the reuse of deallocated memory as *object invalidating sanitizers*. While all of these sanitizers invalidate the memory of deallocated objects in some form, only some permit the eventual reuse of the invalidated memory. The others prohibit reuse indefinitely, creating *one-time allocation* schemes that *prevent* temporal memory safety violations rather than detecting them. The techniques of dangling pointer tagging and lock-and-key are employed by *pointer invalidating sanitizers* to ensure that dangling pointers are unusable.

2.2.1. Object Invalidating Sanitizers. Location-based sanitizers can either invalidate deallocated memory for a certain duration to delay its reuse [8], [9], [11], [12], [14], [15], [16], [18], [19], [20], [21] or invalidate it indefinitely, thereby creating one-time allocation schemes [8], [10], [13], [45], [46], [47]. As mentioned in Section 2.1.1, the only new location-based sanitizers since Song et al. are Scudo [11] and FreeGuard [12]. Both leverage the property of grouping same-sized objects in memory pools to invalidate deallocated memory, with Scudo maintaining the state of memory chunks as embedded metadata within the pool and FreeGuard keeping a list of free memory chunks per pool. Identity-based sanitizers [32], [43] can similarly invalidate the identity of deallocated objects, effectively reverting to the capabilities of location-based sanitizers for temporal memory bug finding. RedFat [32] replaces deallocated objects with red-zones, while EffectiveSan [43] replaces the type metadata of deallocated objects with a special type that consistently results in a type error.

With FFmalloc [45], DangZero [46], and PUMM [47], the use of one-time allocations has seen a resurgence in recent years. As invalidating memory indefinitely can lead to memory starvation in memory-intensive programs, sanitizers must find ways to either reduce memory overhead or limit the scope of one-time allocations. To prevent pointers to small objects from holding onto an entire freed page, FFmalloc allows small objects to share a page, releasing it only when all the objects are freed. DangZero does not strictly enforce one-time allocations and reuses the memory of freed objects once its alias reclaimer—a garbage

collector-style mechanism that pauses the program to scan for dangling pointers—determines that no pointers to the freed objects exist. PUMM does not enforce one-time allocations globally but instead at the scope of an execution unit, which is derived from the program's control-flow graph.

2.2.2. Pointer Invalidating Sanitizers. Object invalidating sanitizers do not detect dereferences of dangling pointers once invalidated memory or identities have been reused for new allocations. To address this, identity-based sanitizers can invalidate pointers directly, as they already maintain metadata per object or per pointer, thereby conceptually achieving complete temporal memory safety. With dangling pointer tagging [36], [41], [48], [49], [50], [51], [52], sanitizers track every pointer to an object, including derived pointers, and invalidate them once the object is deallocated. Since Song et al., CAMP [36] and MarkUs [52] have been introduced, with CAMP simply monitoring pointer creations to maintain its per-object list of valid pointers. MarkUs, however, takes a different approach: instead of tracking pointers in per-object metadata, it tracks freed objects in a quarantine and periodically performs a live-object traversal of accessible memory, similar to a garbage collector, to identify and free quarantined objects without pointers.

Identity-based sanitizers using per-pointer metadata can choose to invalidate pointers using lock-and-key [30], [31], [33], [35], [38], [39], [44], [53], [54]. This involves storing unique identifiers (*keys*) at specific memory locations (*locks*) for each object, and storing the key and the location of the lock for each pointer. At pointer dereferences, the key is checked against the lock to determine the object's state, and upon object deallocation, the lock is nullified, invalidating the key and consequently every pointer. Since the publication of CETS [53], the best-known and conceptually most mature representative of this technique, pointer invalidation using lock-and-key has seen little innovation. However, recent advancements have implemented this technique using pointer tagging, allowing per-object bounds trackers to also utilize lock-and-key. CUP [30] and HWASAN [31] introduced the use of their tag as a key while storing the lock within their object metadata. PACMem [33], CryptSan [34], MTSan [35], and PTAuth [54] leverage ARM hardware features to enhance performance and, in the case of PACMem, CryptSan, and PTAuth, even create cryptographically secure tags.

3. Evaluation Methodology

The primary classification for security-related vulnerabilities is the CWE [64] database, which provides an extensive taxonomy of software and hardware weaknesses. Other significant sources include the systematization of exploit mitigations by Szekeres et al. [1] and the systematization of code sanitization by Song et al. [7], both of which model C/C++-related vulnerabilities as the basis for their analyses. However, while these three taxonomies encompass a wide range of weaknesses and vulnerabilities, they lack precision in defining the characteristics of individual memory bugs. Although the CWE database contains hundreds of weakness categories, only 11 directly relate to memory safety. Since CWE categories are more generic and intended to cover all types of weaknesses, these 11 categories naturally conflate several distinctions among memory bugs that are crucial for differentiating sanitizer detection capabilities. Similarly, Szekeres et al. and Song et al. only briefly address memory safety vulnerabilities. They overlook distinctions between linear and non-linear bugs and do not account for misuse-of-free bugs. Additionally, Song et al. do not consider double-free bugs nor distinguish between overflows and underflows, while Szekeres et al. overlook distinctions between inter-object and intra-object bugs.

To evaluate and compare the capabilities of memory safety sanitizers effectively, it is crucial to identify all relevant memory bugs and categorize their various forms. Consequently, we have developed a new taxonomy of memory bugs that is sufficiently precise to highlight the differences among sanitizers. It is important to note that our focus is exclusively on memory safety bugs. We do not aim to test for specific exploit outcomes (e.g., control-flow hijacking) and do not consider bugs related to the use of uninitialized memory or type confusions that are not directly related to memory safety.

3.1. Spatial Memory Bugs

Spatial memory bugs lead to out-of-bounds accesses (OOBAs), where memory outside the allocated bounds of an object is illegally accessed for *reading* or *writing*. Our categorization, as shown in Figure 1, distinguishes three types of spatial memory bugs: *linear OOBAs*, *non-linear OOBAs*, and *type confusion OOBAs*. OOBAs can either *overflow* an object beyond its upper bound, or *underflow* it below its lower bound. They can further be linear, meaning they are contiguous in memory from one object to another, or non-linear, indicating a direct offset of a pointer into another object. Linear OOBAs commonly arise from mistakes in loop conditions when parsing buffers or errors in size parameters when calling standard library functions such as `memcpy`. Non-linear OOBAs are typically caused by indexing errors. Both linear and non-linear OOBAs can occur within the fields of the same compound object (*intra-object*) or between distinct objects (*inter-object*). A specific case of linear OOBAs is the *non-object* OOBA, where unallocated memory adjacent to an object is illegally accessed. Non-object OOBAs are particularly interesting as they allow for differentiation of sanitizers in terms of padding or alignment issues. Lastly, because C and C++ lack type safety—meaning they do not necessarily perform type checks—accessing memory through a cast to a broader type can also lead to overflows. Type confusion OOBAs, similar to linear OOBAs, can be inter-object, intra-object, and non-object overflows, but not underflows. Type confusion OOBAs test the ability of sanitizers to accurately track the allocated sizes of objects after arbitrary casts.

In Figure 1, we also provide a mapping from our categorization to the corresponding CWE categories. This

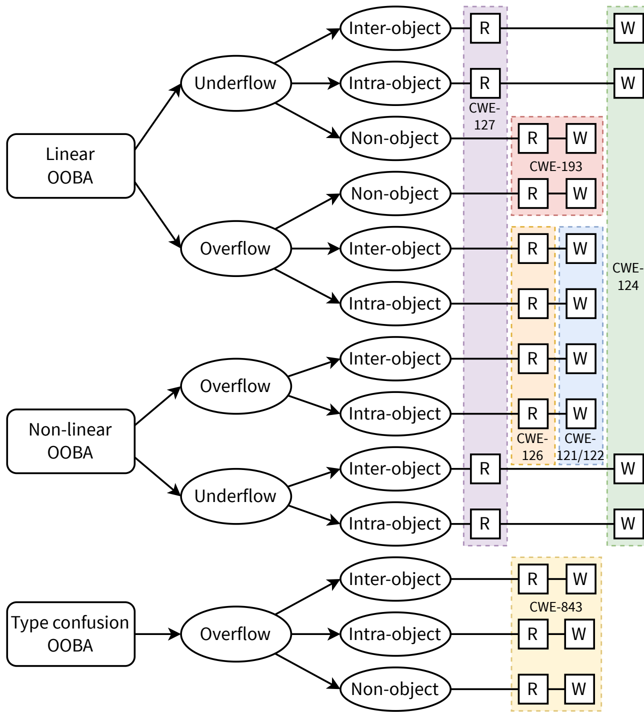

Figure 1. Categorization of spatial memory bugs

mapping reveals that CWE categories group multiple bug types together, rendering them too imprecise for adequately differentiating sanitizer capabilities. More specifically, they do not differentiate between linear and non-linear OOBAs, intra-object, inter-object, and non-object OOBAs, nor, in most cases, between reading and writing.

3.2. Temporal Memory Bugs

Temporal memory bugs refer to accesses to memory that has already been deallocated. In our categorization, as shown in Figure 2, we identify three types of temporal memory bugs: *use-after-∗*, *double-free*, and *misuse-of-free*. Use-after-∗ bugs involve accessing heap objects after their memory has been released (i.e., *use-after-free*), or stack objects when their address has escaped their function (i.e., *use-after-return*) or their scope (i.e., *use-after-scope*). In both cases, the illegally accessed memory can either still be *free* (i.e., deallocated and not yet newly allocated) or already *reused* for other objects. Double-free bugs occur when `free` is called twice on the same pointer, which can cause the allocator to return that same pointer for the next two calls to `malloc`, thereby returning the address of an already *used* object on the second call. Lastly, misuse-of-free bugs occur when pointers not previously returned by `malloc` are passed to `free`. Misuse-of-free bugs can cause the allocator to return specific addresses to still *free* memory or currently *(re)used* memory for subsequent calls to `malloc`.

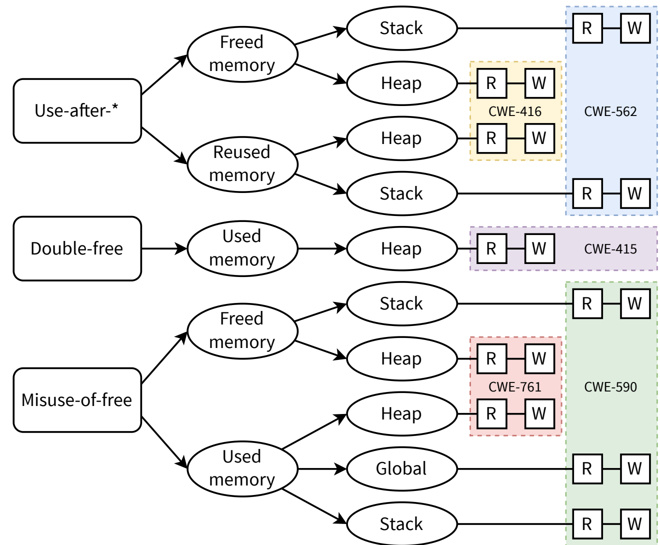

Figure 2. Categorization of temporal memory bugs

In Figure 2, we again provide a mapping from our categorization to CWE categories, which illustrates the conflation in the CWE classification. Most notably, the CWE classification does not differentiate between corruptions of used, freed, and reused memory, nor does it distinguish between reading and writing for any type of temporal memory bug. Furthermore, misuse-of-free bugs are only assigned two categories: one for heap pointers and another for both stack and global pointers.

3.3. Test Case Primitives

To translate our memory bug taxonomy into our evaluation tool, we deconstruct memory bugs into three fundamental properties: the *memory region* in which the vulnerable object resides, the *type of bug* causing the memory corruption, and the *type of access* gained to the object. For each property, we define distinct *primitives* from which actual memory corruptions—and later test cases—can be formed. Figure 3 provides an overview of these primitives and illustrates possible combinations for forming actual corruptions.

Memory Region. Depending on the type of memory bug, there can be two different memory regions involved. The first refers to the region where the memory bug takes effect and where the corruption occurs—that is, the region where the *target object* resides. For temporal bugs, the target region alone is sufficient, as temporal bugs only involve a single object. For example, use-after-free bugs only target heap objects by definition. However, for spatial bugs, there are typically two objects involved: one that is overflowed or underflowed, and another that is illegally accessed through the overflow or underflow. Thus, we specify a second memory region for spatial bugs where the memory bug originates—that is, the region of the *origin object* being overflowed or underflowed.

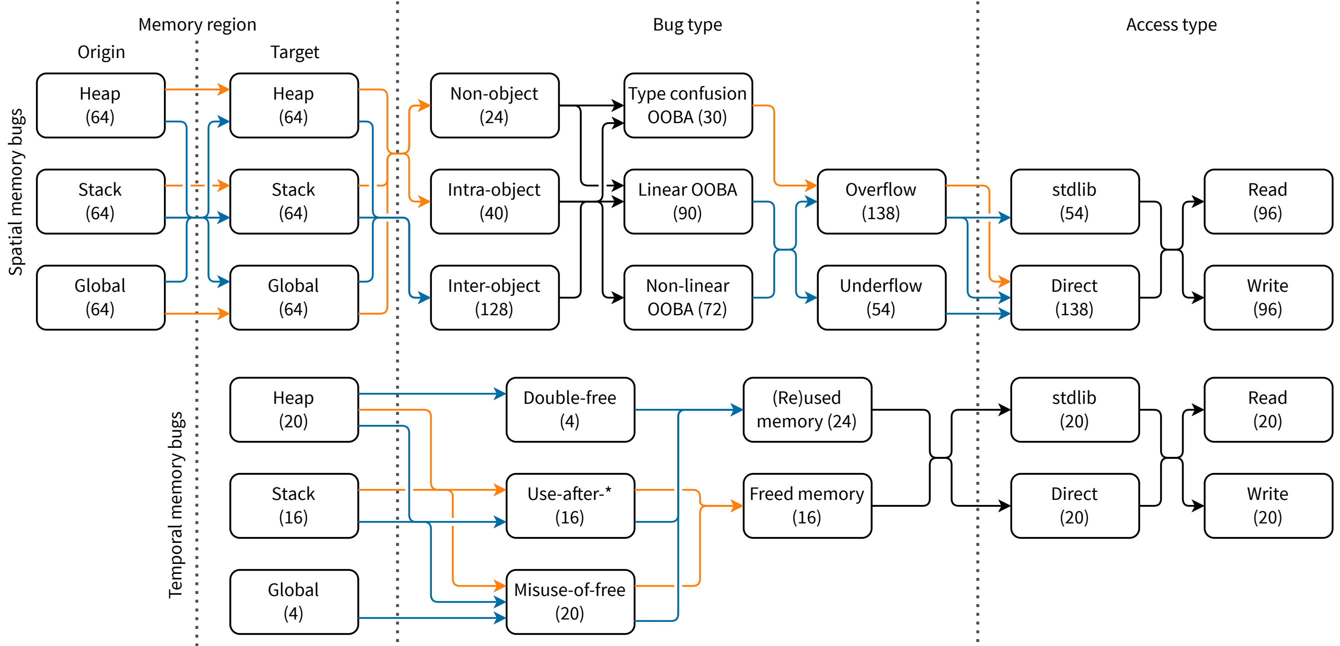

Figure 3. Possible combinations of primitives for generating the 232 test cases. The number of combinations containing each primitive is shown in parentheses. Colored input arrows are connected with the outputs of the same color.

For both types of memory regions, we define their primitives based on the traditional memory layout of programs: the *heap* for dynamically allocated objects via the standard library (using `malloc`[2]), the *stack* for local function variables, and *global* memory for variables with global scope and program-long lifespan. For spatial bugs, we denote the memory region as an (origin, target) tuple. For example, an overflow of a stack buffer that corrupts a heap object is represented as (stack, heap), while a traditional fully stack-based buffer overflow is denoted as (stack, stack). In most combinations, target and origin are identical, but this differentiation allows us to assess a sanitizer's effectiveness in protecting memory regions from one another.

Bug Type. The primary characteristic of a memory corruption or leak is the bug type. We define the primitives for this property directly as categorized before and shown in Figures 1 and 2. For temporal bugs, the primitives include use-after-∗ and misuse-of-free, both targeting (re)used and freed memory, and double-free, which always targets reused memory. For spatial bugs, the primitives include linear, non-linear, and type confusion OOBAs. These are further subdivided according to Figure 1 into inter-object, intra-object, and non-object OOBAs, and then into underflows and overflows. The constraints from Section 3.1 apply to their respective bug types: non-linear OOBAs are incompatible with non-object accesses, and type confusion OOBAs cannot be used for underflows.

2. We do not use `mmap` as an additional method for dynamic allocations, since it is typically used for specific purposes, such as custom allocators, and would inflate our test cases disproportionally.

Access Type. The access type refers to whether a memory corruption (*write* primitives) or memory leak (*read* primitives) occurs, and whether it occurs directly in the code (*direct* primitives) or through a standard library function (*stdlib* primitives), such as `memcpy`. Investigating bugs triggered via standard library functions is crucial for our evaluation tool, as these functions are often not compiled with sanitizers for practical reasons and remain unprotected unless the sanitizer explicitly secures relevant calls, for example, by wrapping them.

3.4. Test Cases

Based on the primitives described above, we form 232 unique test cases for our evaluation tool. Each test case utilizes a specific memory bug type—that is, a combination of bug type primitives—to access a target object—that is, a combination of memory region and access type primitives. Figure 3 illustrates the possible combinations. In the case of spatial bugs, intra-object and non-object OOBAs are restricted to single objects, which requires the memory regions of the origin and target to be identical (e.g., from heap only to heap). Furthermore, type confusion OOBAs cannot be combined with underflows or with accesses through the standard library. Regarding temporal memory bugs, global variables are only part of misuse-of-free bugs since they cannot be deallocated during runtime. In addition to global variables, stack variables cannot be targets of double-free bugs. In total, this results in 192 meaningful combinations for spatial bugs and 40 for temporal bugs.

LISTING 1. EXAMPLE TEST CASE THAT CORRUPTS A STACK TARGET FROM THE STACK USING A NON-LINEAR UNDERFLOW OOBA

```
// Target region: stack
char target[4] = "abc";
// Origin region: stack
char origin[4];
// Bug type: non-linear underflow OOBA
if (!(target < origin)) return PRECOND_FAILED;
size_t distance = origin - target;
for(size_t i = 0; i < 4; i++)
  // Acces type: write
  origin[i - distance] = 0xFF;
return TEST_CASE_SUCCESSFUL;
```

LISTING 2. TEMPLATE FOR THE STACK TARGET PRIMITIVE

```
target.region=STACK
target.code = "char␣target[4]␣=␣\"abc\";"
```

4. Evaluation Tool

In accordance with our evaluation methodology, we developed the Memory Sanitizer Evaluation Tool (MSET). This tool combines memory region, bug type, and access type primitives to create small, self-contained C programs as test cases. MSET compiles these test cases with each sanitizer under evaluation, executing them sequentially and assessing their outcomes based on their exit status. Although MSET is written in modern C++, requiring a modern compiler, the code it generates can be compiled and executed on any system that supports standard C. This allows MSET to support various platforms, making it particularly valuable for evaluating hardware-specific sanitizers. We have tested the tool and its generated test cases on x86-64 Debian/Ubuntu Linux and ARM64 Ubuntu Linux.

The generated test cases are concise and straightforward, enhancing the tool's adaptability for evaluating both existing and new sanitizers, including proof-of-concept implementations with minimal support for C/C++. The simplicity of the generated code also enables users to quickly analyze undetected memory bugs, which typically indicate false negatives in the sanitizer. While the test cases may not always reflect the complexity of real-world programs, their simplicity allows MSET to establish an upper bound on the capabilities of sanitizers. If a sanitizer fails a basic test, it is highly likely to also fail a more complex variant of the same test.

4.1. Test Case Generation

As detailed in Section 3.4, MSET creates 232 unique test cases designed to contain intentional memory bugs. To facilitate this, MSET provides a suite of templates for all primitives and combines them to form the test cases. Listing 1 presents an example of a test case that attempts to corrupt a stack target from a stack origin using a non-linear underflow OOBA. Listings 2 to 4 display the templates used to generate this specific test. Listing 2 shows a simplified version of the template for a stack target region. The generated code from this template is straightforward:

LISTING 3. TEMPLATE FOR THE PRIMITIVE COMBINATION OF A NON-LINEAR UNDERFLOW ORIGINATING FROM THE STACK

```
bug_type.origin_region=STACK
bug_type.origin_code="char␣origin[4];"
bug_type.precond_check="&$TARGET[0]␣<␣origin"
bug_type.code=
  "size_t␣distance␣=␣$ORIGIN␣-␣&$TARGET[0];",
  "for(size_t␣i␣=␣0;␣i␣<␣4;␣i++)"
bug_type.access_index = "i␣-␣distance";
```

LISTING 4. TEMPLATE FOR THE WRITE PRIMITIVE

```
access_type.code = "$ORIGIN[$INDEX]␣=␣0xFF;"
```

it involves allocating a stack variable and its initialization. Listing 3 displays the template for a non-linear underflow from a stack origin, which includes the code for allocating the buffer and the code that leads to memory corruption. Finally, Listing 4 presents the template for the actual write access that corrupts the memory.

For each test case, MSET also generates a bug-free version intended to detect *false positives* caused by sanitizers. MSET verifies whether the tested sanitizer can compile and run the test case without its memory bug while retaining the features utilized by the original test case. Only if this bug-free version compiles and runs successfully, the actual test case is compiled and executed. A failure in the bug-free version typically indicates a false positive caused by the sanitizer, thereby aiding developers in identifying deficiencies in their sanitizer implementations.

Moreover, MSET's test case generation considers various implementation details to ensure sound test results. For spatial memory bugs, loop unrolling is used to eliminate the need for additional variables that could be inadvertently corrupted, potentially masking the memory bug. To avoid false positives for temporal memory bugs, calls to the standard library are avoided, as they may allocate memory and affect the results. When a test case requires the use of standard library functions, `memcpy` is used for reading, and `memset` is used for writing. Heap objects are allocated using `malloc` and deallocated using `free`. Our misuse-of-free and double-free test cases are specifically tailored for *glibc*. For custom allocators, they still produce the correct basic results, correctly indicating a failure to detect invalid pointers being freed, but offer no further differentiation regarding the memory type (see also Section 6).

4.2. Test Case Variants

In addition to good and bad versions of test cases, MSET generates different *variants* of certain test cases. Variants define approaches to trigger the same memory bug, i.e., the specific combination of primitives defined by the test case, under different conditions. An example of such a condition is the relative ordering of origin and target objects. A test case might be infeasible at runtime if an overflow originating from the stack attempts to target another stack object if the sanitizer places the target lower than the origin in memory. To prevent MSET from incorrectly identifying this test case

as detected by the sanitizer, it deploys test case variants that attempt different orderings of the origin and target objects at runtime.

When MSET performs its evaluation, it executes all available variants of a test case until *the first one* succeeds or *none of them* do. As soon as the first variant succeeds, MSET considers the test's memory bug undetected by the sanitizer and proceeds to the next test case. Consequently, individual variants do *not* directly count towards the evaluation result of the sanitizer; rather, they serve as a means to achieve the same memory corruption under varying conditions. Thus, variants are an essential tool that ensures the robustness of MSET and the comparability of its evaluation results.

Other applications of variants include the placement of auxiliary variables and different approaches to trigger certain memory bugs. Auxiliary variables may be used for linear OOBAs, requiring different variants to prevent their accidental corruption when positioned between the origin and target objects by the sanitizer (or compiler). For type confusion OOBAs, MSET employs two variants: one that casts a pointer to the base of the origin object to a type large enough to reach the target in multiple accesses and another that casts a pointer to the last byte of the origin to a base type that requires a single, word-sized access. For double-free bugs, MSET tests different sequences in which the target object and sometimes auxiliary dummy objects are freed; for misuse-of-free bugs, MSET tries different "magic" values; and for use-after-return bugs, MSET explores different stack frame layouts to achieve its goal.

4.3. Test Case Execution

MSET executes test cases one by one and records whether the sanitizer successfully detected the memory corruption for each test case. If the memory bug of a test case is successfully triggered, the test case returns a *success* status, indicating that the sanitizer failed to detect the memory corruption. If the memory corruption is detected by the sanitizer, the test case returns an *error* status or raises a *segmentation fault*. We use this approach since some sanitizers intentionally cause segmentation faults for invalid accesses using red-zones or guard pages (refer to Section 2). As discussed above, test case variants may be deemed infeasible at runtime, in which case a special *failing precondition* status is returned. If all variants of a test case return with this status, MSET considers the memory corruption as prevented, since some sanitizers may rearrange the memory layout for security reasons. Lastly, if the test case returns *any other* status, MSET conservatively considers the memory corruption also as prevented. Note that, as mentioned before, for test cases with multiple variants, MSET considers the corresponding memory bug as detected only if *all* test case variants are unsuccessful, meaning none of them return the success status.

5. Sanitizer Evaluation

To assess the effectiveness of existing memory safety sanitizers, we utilize MSET to evaluate 16 different sanitizers. Note that ASan [18] and ASan-- [21] are evaluated together, as are SoftBound [40], CETS [53], and their recent revision [44]. The complete list of considered sanitizers is provided in Table 1. For details on the specific version of each sanitizer, the systems used for testing, and the reasons for excluding 10 sanitizers with available online implementations, please refer to Section A of the Appendix. To ensure comparability of results across the 232 test cases, we categorize them based on the six bug types: linear, non-linear, and type confusion OOBAs, as well as use-after-$*$, double-free, and misuse-of-free bugs. We report and discuss results for each of the six bug types individually, as summarizing them into an overall score requires weighting the bug types by severity and relevance, which can change over time and is prone to bias. To visualize the results, we employ 6-dimensional radar charts, with each axis representing the percentage of test cases that were successfully mitigated for the respective bug type—essentially, the bug *detection rate*. Each axis thus encompasses all possible combinations of primitives for that bug type.

5.1. Conceptual Potentials

Depending on the chosen sanitizing technique(s), memory safety sanitizers exhibit varying capabilities for detecting each of the six different bug types. Consequently, the effectiveness of a sanitizer in detecting specific memory bugs can be constrained by conceptual factors. To contextualize the detection rates observed in our evaluation, we assess the theoretical maximum detection rate—the *conceptual potential*—for each sanitizing technique. Figure 4 displays the conceptual potentials of the seven sanitizing techniques introduced in Section 2. Since half of the evaluated sanitizers only protect heap memory, we present their conceptual potentials separately in the figure. When employing spatial sanitizing techniques, these sanitizers can only detect memory bugs that target or originate from the heap, thus achieving only one-third of the potential of their full memory-protecting counterparts. Similarly, when using temporal sanitizing techniques, they can only achieve half of the potential for use-after-$*$ bugs, detecting only bugs on the heap and not on the stack.

For linear and type confusion bugs, inter-object OOBAs account for 60% of the total number of test cases, as they can occur between objects in the same memory region and between objects in different regions. Non-object OOBAs and intra-object OOBAs, on the other hand, are only possible within the same memory region, each comprising 20% of the total. For the non-linear bug type, non-object OOBAs are not considered. Consequently, 75% of the non-linear test cases correspond to intra-object OOBAs, while 25% correspond to inter-object OOBAs.

Location-based sanitizers cannot detect any intra-object OOBAs but can detect inter-object linear and type confusion

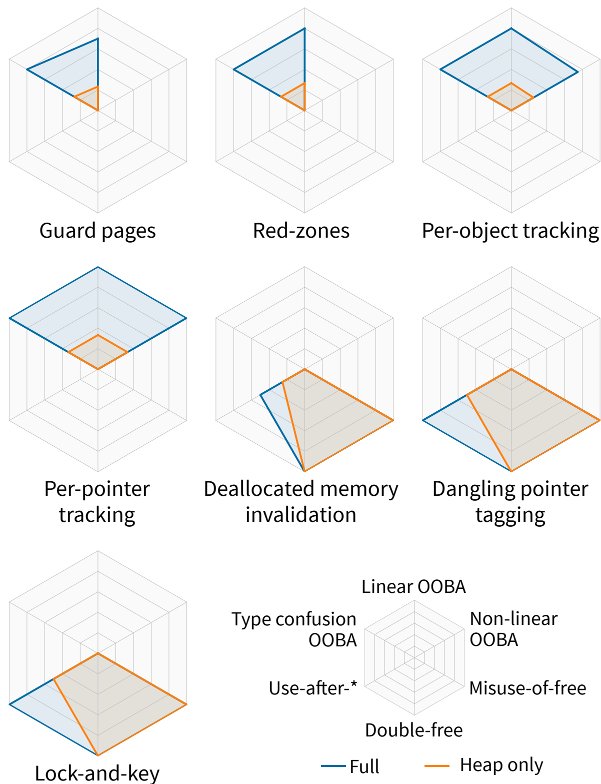

Figure 4. Conceptual detection potential for each sanitizing technique.

OOBAs. For the latter two bug types, only 50% of non-object OOBAs are detectable when using guard pages, as objects are typically placed either directly before or after a guard page, introducing padding on the opposite side of the object. Therefore, for sanitizers protecting stack and global memory in addition to heap memory, employing guard pages can achieve detection rates of up to 70% for linear and type confusion OOBAs (60% corresponding to intra-object and 10% to non-object OOBAs, i.e., half of them). Sanitizers can enhance their detection capabilities to 80% by implementing red-zones and detecting all non-object OOBAs. However, both techniques remain unable to detect the remaining 20% of linear and type confusion OOBAs, which are intra-object, and all non-linear OOBAs.

Identity-based sanitizers employing per-object tracking can detect, in addition to 80% of linear and type confusion OOBAs, up to 75% of non-linear OOBAs, missing only intra-object OOBAs. Such intra-object OOBAs are only detectable by sanitizers that track bounds per pointer. These sanitizers have the potential for complete spatial memory safety, with detection capabilities of up to 100% for all types of spatial bugs.

Regarding temporal memory safety, sanitizers that invalidate deallocated memory can detect double-free and misuse-of-free bugs, but can only detect use-after-$*$ bugs on freed (and not yet reallocated) memory. Consequently, their conceptual potential is limited to at most 50% for use-after-$*$ bug types. Dangling pointer tagging and lock-and-key sanitizers can additionally detect use-after-$*$ bugs on reused memory, enabling them to achieve detection rates of up to 100% for all temporal bug types.

5.2. Evaluation Results

Figure 5 presents the evaluation results for the 16 sanitizers. For those employing randomization, specifically FreeGuard [12], HWASAN [31], and Scudo [11], we utilize the arithmetic mean derived from 10 evaluation runs. The underlying data for the percentages shown in the radar charts can be found in Appendix B, along with the standard deviations for the results of the randomizing sanitizers. Due to the intentional simplicity of MSET's test cases, all 16 sanitizers successfully compiled and executed their bug-free versions without triggering false positives.

Modern systems and toolchains can prevent certain memory bugs even in the absence of sanitization. To account for this, we establish a *baseline*: for each sanitizer, we compile and execute all test cases in an identical setup *without* the sanitizer. The baseline results, which are overlaid in the radar charts, are relevant because certain sanitizers can introduce changes that may lead to false negatives within the baseline. Such changes may include replacing the standard allocator and weakening its built-in protections, or altering the program's memory layout. An example of the latter is FreeGuard, which allows non-linear underflows from global memory to the heap. To emphasize discrepancies between the conceptual potential and the actual bug-finding capabilities of the tested sanitizers, each radar chart is further overlaid with the sanitizer's conceptual potential. Differences between the conceptual potential and the actual evaluation result may indicate issues within the sanitizer, such as an incomplete implementation or errors in applying its sanitizing technique(s) correctly. Note that for heap-only sanitizers, the figure reflects the heap-only potential. To account for the baseline detection capability, it is added to the displayed potential of each sanitizer in Figure 5. Below, we provide further details on the baseline and discuss the evaluation results for each sanitizer.

The baselines observed in our evaluation show detection rates of 40% for linear and type confusion OOBAs, 25% for non-linear OOBAs, and 0% for all other bug types. Moreover, the baselines are consistent across all sanitizers, meaning the results for their respective test setups without sanitization are identical. Out of the 232 test cases, 166 successfully triggered their memory bugs, while 42 had unmet runtime preconditions and 24 produced segmentation faults. The standard memory layout of Linux systems, which places unmapped memory pages between the heap, stack, and global memory, led the Linux kernel to fault on the test cases involving linear or type confusion OOBAs across different memory segments. The test cases that failed to meet runtime preconditions involved any OOBAs between different memory segments where the target buffer was unreachable from the origin buffer, such as in a stack-to-heap overflow when the heap is located below the stack.

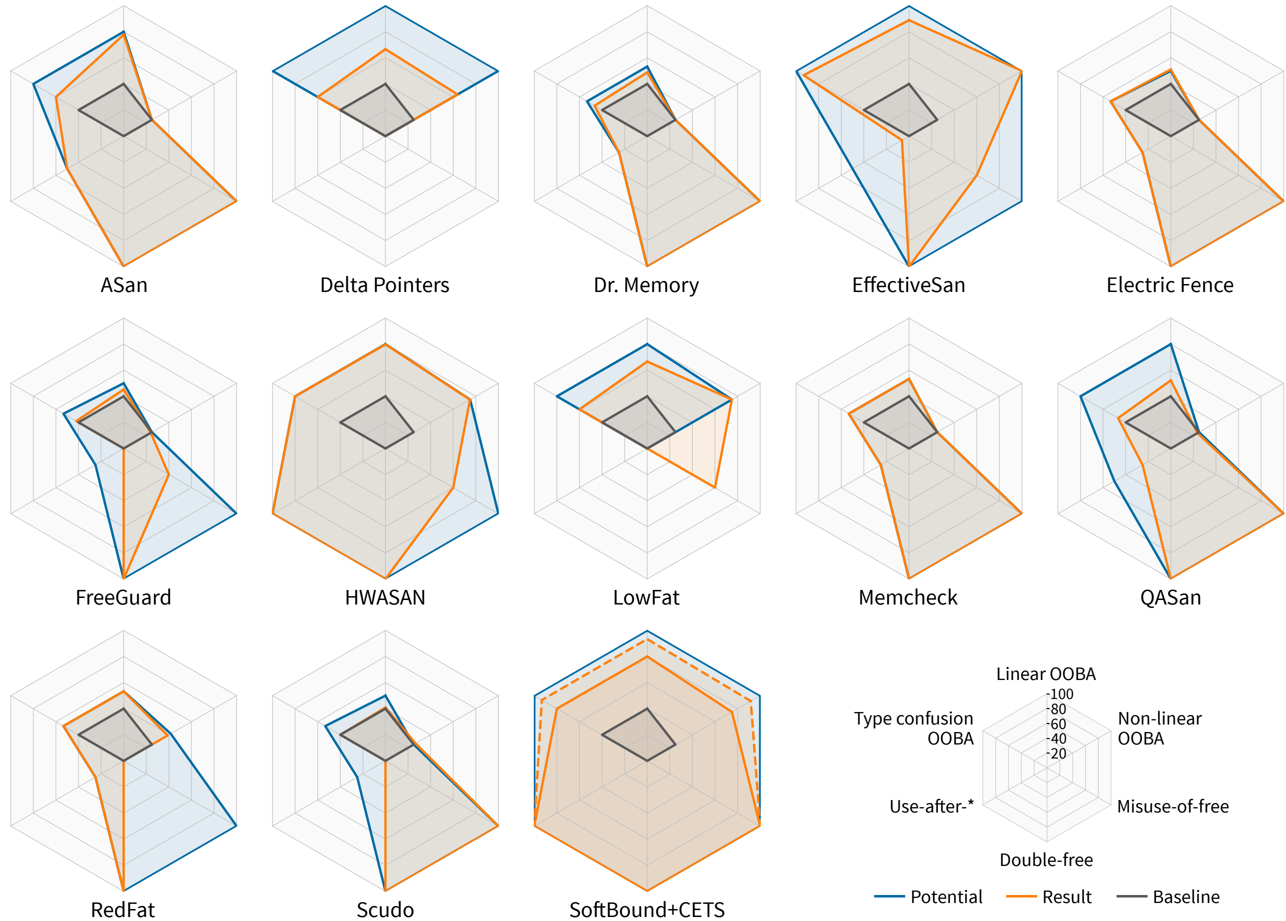

Figure 5. Overview of sanitizers evaluation results summarized by bug type.

SoftBound+CETS [40], [53] offers protection for the heap, stack, and global memory by tracking object bounds per pointer and locking pointers to objects with keys. While this approach should conceptually ensure complete memory safety and CETS indeed detects 100% of temporal bugs, SoftBound only detects 80% of linear and type confusion OOBAs and 75% of non-linear OOBAs. We found that all missed memory bugs stem from SoftBound's lack of protection against intra-object OOBAs. Notably, SoftBound is conceptually capable of tracking intra-object bounds: when a pointer to an object field is derived, the per-pointer bounds metadata should narrow to reflect the field's bounds. However, the original implementation of SoftBound does not support this feature. This is not reflected in the evaluation presented in its original paper or in the conceptual review provided by Song et al. [7].

SoftBound has recently seen a revision [44] that adds support for detecting intra-object OOBAs. Our results for this update, shown with a dashed outline in Figure 5, reveal that while all stack and heap-located intra-object OOBAs are detected, none of the global ones are, resulting in detection rates of 93.3% for linear OOBAs (the highest among the evaluated sanitizers), 93.3% for type confusion OOBAs (shared highest), and 91.7% for non-linear OOBAs. This shortcoming is not reflected in the project's Juliet Test Suite [57] evaluation, as it does not contain intra-object global OOBAs.

LowFat [27], [28], [29] is a spatial memory safety sanitizer that tracks bounds information per object. It segregates objects into dedicated memory regions based on their size, thereby creating an implicit encoding of their bounds within their location. LowFat fully utilizes this technique to protect against all inter-object OOBAs. However, as a per-object tracker, it cannot protect against intra-object OOBAs. Additionally, due to fixed object sizes and resultant padding, in which overflows remain undetected, it fails to prevent non-object overflows. Nevertheless, it detects all non-object underflows since padding is only added after objects. Despite its use of low-fat pointers [65], LowFat can detect all non-linear inter-object OOBAs, because it tracks bounds per pointer within functions and instruments pointer arithmetic when a newly derived pointer is set to leave a function. Overall, LowFat achieves detection rates of 66.7% for linear, 75% for non-linear, and 60% for type confusion OOBAs.

As a spatial memory safety sanitizer, LowFat does not aim to protect against temporal memory bugs. However, it does implement its own heap allocator wrappers that can detect misuse-of-free errors on stack and global pointers with a detection rate of 60%.

EffectiveSan [43] combines the low-fat pointer technique with type checking to track object bounds per pointer. Additionally, it protects against temporal memory errors by associating deallocated objects with a special type. However, our evaluation shows that EffectiveSan misses all use-after-∗ bugs on reused memory. This is expected since MSET's test cases use the same type for reallocated objects—a behavior not uncommon in real programs that goes unnoticed by EffectiveSan's type-based detection. Moreover, most use-after-∗ bugs on freed memory are also missed, resulting in a detection rate of 6.25% for use-after-∗ bugs in general. This is a known limitation of EffectiveSan: although it associates deallocated objects with a special type, the necessary type check is not always performed. Due to this issue, EffectiveSan only detects use-after-free bugs on heap objects when used for direct writing (i.e., not using the standard library for the access). Nevertheless, it detects 100% of double-free bugs and 60% of misuse-of-free bugs, missing only those bugs where a pointer passed to `free` points to the heap but not to the start of an allocated object (i.e., CWE-761 [64]). Since EffectiveSan already tracks the beginnings of objects, the missing check could be easily added, and we have notified the maintainers.

EffectiveSan is the only sanitizer, besides the SoftBound+CETS revision [44], that detects some intra-object OOBAs, achieving detection rates close to its 100% potential as a per-pointer sanitizer. It detects 88.9% of linear OOBAs, missing half of the non-object OOBAs and one-third of the intra-object OOBAs. The undetected intra-object bugs occur when casting to char pointers, which are always treated as pointers to the whole object by EffectiveSan. Additionally, when copying from global objects via the standard library, EffectiveSan does not consider the size argument in its `memcpy` checks, leading to undetected non-object overwrites. We have reported this to the maintainers of EffectiveSan and they have confirmed that `memcpy` is not considered during instrumentation. For type confusion bugs, EffectiveSan achieves a detection rate of 93.3%, missing only intra-object OOBAs on global objects, and shares the highest detection rate among the evaluated sanitizers with the SoftBound+CETS revision. It is the only sanitizer to successfully detect 100% of non-linear OOBAs, including those that are intra-object. We attribute the missed bugs to the prototype state of EffectiveSan's implementation and the design decisions regarding casting to char pointers.

RedFat [32] is another sanitizer employing the low-fat pointer technique. RedFat utilizes both low-fat pointers (for per-object tracking) and red-zones to protect heap objects against spatial bugs. To detect temporal bugs, it employs the deallocated memory invalidation technique. As a heap-only sanitizer, RedFat does not detect memory errors on the stack and global memory. Moreover, its per-object concept does not enable it to detect any intra-object OOBAs. In terms of linear and type confusion OOBAs, its detection rates reach their potential of 53.3%. As RedFat uses low-fat pointers only for heap objects it detects only non-linear OOBAs that originate on the heap and not those that target it from unchecked, normal pointers, resulting in a detection rate of 38.9% out of a possible 41.7%. Similar to LowFat, it detects non-linear inter-object OOBAs by instrumenting pointer arithmetic.

Regarding temporal safety, RedFat detects 100% of double-free bugs but none of the misuse-of-free bugs. Since RedFat uses a custom allocator, it could be extended to check the pointers it receives, potentially improving the misuse-of-free detection rate to 100%. In response to our issue report, the maintainers confirmed that not checking non-RedFat pointers is intentional and necessary for compatibility reasons. In addition, RedFat does not check heap pointers that were not returned by `malloc`, for which we have submitted a second issue report. Due to its deallocated memory invalidation technique, RedFat cannot detect use-after-∗ bugs on reused memory. However, it successfully detects all use-after-∗ bugs on freed heap memory, resulting in a detection rate of 25% for the use-after-∗ category.

ASan [18] provides spatial and temporal memory safety for heap, stack, and globals by tracking bounds per object and invalidating deallocated memory. Widely utilized in real-world applications, it is available for both Clang and GCC. Our evaluation did not reveal any differences between these implementations. For spatial safety, ASan places red-zones around objects to prevent linear and type confusion OOBAs. However, using red-zones, it cannot detect non-linear and intra-object OOBAs. Additionally, ASan fails to detect non-object linear underflow reads and writes for global objects, an implementation issue we have reported to its maintainers. According to them, red-zones are only placed *after* global objects. Therefore, the first global object will always allow for non-object underflows. The variants of type confusion OOBAs into non-objects that use unaligned load widening remain undetected, a documented limitation of ASan, which is accepted in favor of performance. Lacking intra-object protection, ASan achieves detection rates of 77.7% (out of an 80% potential) for linear OOBAs and 60% (out of an 80% potential) for type confusion OOBAs.

For temporal safety, ASan invalidates deallocated memory and delays its reuse to detect double-free, misuse-of-free, and use-after-∗ errors on freed memory. However, since ASan does not strictly enforce its memory reuse delay and allows programs to specifically allocate memory at reused addresses, MSET's use-after-∗ test cases on reused memory are unaffected by the delay, resulting in all tests on reused memory going undetected and a detection rate of 50%, reaching the potential for deallocated memory invalidation sanitizers. Aside from the undetected linear and type confusion OOBAs mentioned earlier, ASan achieves its full potential, demonstrating the maturity of its implementation.

ASan-- [21] is an optimized version of ASan that aims to retain ASan's capabilities while increasing its performance. Our evaluation shows that ASan-- successfully achieves the same detection rates as ASan, inheriting also its limitations.

QASan [19] implements ASan's algorithm but operates on binaries and only protects the heap. In terms of spatial safety, QASan reaches its potential of 53.3% for type confusion OOBAs. For linear and non-linear OOBAs, it almost reaches its potential, achieving detection rates of 52.2% and 23.6%, respectively. ASan, when protecting only the heap, has slightly better detection rates of 53.3% and 25% for the two categories. QASan fails to detect a heap non-object overflow using the standard library, due to a bug that we have reported to the authors. Moreover, its modified heap layout allows for linear OOBAs from the heap to the global data section, which are detected in the baseline, to escape detection. QASan successfully detects all use-after-$*$ errors on freed heap memory (25% detection rate) and all double-free and misuse-of-free errors. In contrast to ASan, attempts to reuse heap memory lead to memory starvation.

HWASAN [31] provides spatial and temporal memory safety for heap, stack, and globals by employing hardware-assisted pointer tagging. It tracks the bounds and liveliness of objects by storing a per-object identifier in a shadow memory, where the identifier occupies space equivalent to the object's size. Additionally, HWASAN utilizes the Top Byte Ignore feature on ARM platforms to hold the identifier in the unused bits of pointers. For spatial safety, HWASAN effectively prevents all inter-object OOBAs, albeit in a probabilistic manner. The probabilistic nature of detection stems from HWASAN's limited number of available tag bits per pointer, which inevitably results in objects sharing the same identifier. Consequently, this allows some memory errors to remain undetected. As a per-object tracking sanitizer, HWASAN does not protect against intra-object OOBAs. It reaches its potential of detecting 80% of type confusion OOBAs but falls 0.2% short of its 80% potential for linear OOBAs and 0.1% short of its 80% potential for non-linear OOBAs. We ascribe this shortfall to the aforementioned tag collisions. It reliably detects 100% of use-after-$*$ and double-free bugs but only 60% of the misuses-of-free bugs, missing those on heap pointers. We have notified the maintainers about this shortcoming.

Delta Pointers [42] employs software pointer tagging to encode bounds metadata directly in pointers, providing spatial memory safety for heap, stack, and global memory. However, despite tracking bounds per pointer, Delta Pointers does not offer protection against intra-object OOBAs. This appears to be a mere implementation issue that should be addressable within its current tagging scheme. By design, Delta Pointers does not check for underflows and considers unaligned load widening accesses outside its threat model. Consequently, it cannot achieve the per-pointer technique's conceptual potentials of 100% for all types of spatial bugs. Delta Pointers detects 66.7% of linear OOBAs and 63.9% of non-linear OOBAs, but it does not detect the variants of type confusion OOBAs that use unaligned load widening, achieving a detection rate of 60% in this category. As a spatial-only sanitizer, Delta Pointers does not cover temporal memory safety.

Dr. Memory [16] and **Memcheck** [15] are heap-only sanitizers that ensure spatial and temporal memory safety by inserting red-zones between heap objects and invalidating deallocated memory. Memcheck additionally aims to detect some stack bugs, such as use-after-scope or overflowing the top of the stack. All detected spatial bugs are inter-object and non-object OOBAs originating in the heap. While Memcheck successfully reaches its potentials of 53.3% for linear and type confusion OOBAs and 25% for non-linear OOBAs, Dr. Memory fails to detect linear underflows between two heap objects and faces challenges detecting non-object OOBAs, achieving detection rates of 48.9% for linear and 46.7% for type confusion OOBAs. Regarding temporal safety, both sanitizers reach their potentials and provide 100% detection of double-free and misuse-of-free errors. They invalidate deallocated memory by placing red-zones over freed memory, preventing all use-after-$*$ errors that target freed heap memory. However, as deallocated memory invalidation does not counteract MSET's tests on reused memory, their detection rate for use-after-$*$ bugs is 25%.

Electric Fence [8], **FreeGuard** [12], and **Scudo** [11] are heap-only sanitizers that ensure spatial and temporal memory safety by surrounding heap objects with guard pages and employing deallocated memory invalidation. As location-based sanitizers, they cannot detect intra-object OOBAs. Scudo sporadically thwarts certain non-linear OOBAs by randomly distributing objects on the heap, rendering some test case variants that overflow or underflow impossible due to the relative positions of the target and origin. FreeGuard additionally randomizes the heap's base address, causing global objects not always to be located below the heap. Consequently, non-linear underflows originating in global memory can successfully overwrite heap objects, which is not possible in the baseline, resulting in a detection rate of 24.6% for non-linear OOBAs. With a detection rate of 25%, Electric Fence matches the baseline for non-linear OOBAs but can detect all linear and type confusion OOBAs between heap objects, reaching its potential and achieving detection rates of 51.1% and 53.3% in these categories. FreeGuard sporadically detects some linear and type confusion heap OOBAs, resulting in rates of 45.2% and 42%, respectively. Scudo relies on randomization to obscure object locations and groups smaller heap objects into regions surrounded by guard pages. Since reaching a heap target from a heap origin in MSET test cases is done within a single region, Scudo can only detect a negligible number of spatial memory errors. It achieves detection rates of 40.0% for type confusion OOBAs and 40.9% for linear OOBAs, which is only marginally better than the baselines of 40%.

Regarding temporal safety, all three sanitizers employ deallocated memory invalidation for the heap and can conceptually only detect use-after-free errors on freed memory. Only Electric Fence reaches its full potential and detects 25% of use-after-$*$ bugs. Scudo and FreeGuard fail to detect any of MSET's use-after-$*$ bugs due to their randomized, best-effort approaches to memory reuse delays. All three sanitizers can conceptually detect 100% of double-free and misuse-of-free bugs, and Electric Fence and Scudo do so in practice. FreeGuard implements the correct checks for detecting all misuses-of-free but achieves only a 40% de-

tection rate in this category because it aborts the execution of the code only when detecting non-heap pointers. We have notified its maintainers of this potential oversight.

5.3. Summary

First, our evaluation assesses and compares the quantitative potential of various sanitization techniques. While existing categorizations of sanitizers and their techniques provide some insights, a quantitative analysis improves our understanding of the completeness of individual techniques and their combinations. It reveals that per-pointer tracking is essential for achieving full spatial memory safety, whereas lock-and-key techniques and dangling pointer tagging can ensure complete temporal safety.

Second, our evaluation shows that, although a few sanitizers realize their full conceptual detection potential, the majority fall short, often due to incomplete implementations or errors in applying sanitizing techniques correctly. Of the 16 evaluated sanitizers, only Electric Fence and Memcheck demonstrate complete and sound implementations of their respective sanitizing techniques, allowing them to fulfill their conceptual potential. Six sanitizers—ASan, ASan--, QASan, RedFat, EffectiveSan, and Delta Pointers—have made design decisions, such as prioritizing performance or compatibility, that prevent them from reaching their conceptual potential. Delta Pointers and SoftBound+CETS have unimplemented features, specifically support for intra-object handling. Implementation bugs or oversights lead to false negatives for FreeGuard, QASan, EffectiveSan, Dr. Memory, and the revised version of SoftBound+CETS. EffectiveSan, FreeGuard, HWASAN, and RedFat share a common oversight (or intentional choice) in handling misuse-of-free bugs. These limitations can be readily addressed for all three sanitizers, and we have notified the authors accordingly. FreeGuard and HWASAN face additional limitations due to their reliance on randomness and LowFat due to the need for object padding. FreeGuard and LowFat can only resolve their limitation by sacrificing performance, while HWASAN cannot easily overcome its limitation due to the hardware-imposed tag size constraints.

Third, a notable trend from our evaluation is that research on memory safety sanitizers is not focusing on enhancing detection capabilities but, as far as we can ascertain, on improving performance, particularly to increase usability in fuzzing. Sanitizers such as PACMem [33], PTAuth [54], MTSan [35], and CryptSan [34] leverage recent hardware advancements to boost their performance. QASan [19], CAMP [36], and PTAuth, among others, are designed to protect only the heap and often only against temporal memory bugs. ASan--, QASan, and RetroWrite [20] build upon the concept of ASan, either to enhance speed or to enable direct application for binaries. Although proven effective in practice, ASan's concept is not complete. We argue that while general usability and integration with fuzzing are crucial for finding real-world bugs and vulnerabilities, a significant proportion of potential memory bugs will remain conceptually undetectable.

In summary, our results demonstrate that relying solely on conceptual evaluations—a common practice in memory safety sanitizer research—overlooks practical implementation challenges and leads to less effective sanitizers in practice. More recent sanitizers attempt to address this issue by utilizing the Juliet Test Suite; however, this approach is insufficient, as evidenced by its sometimes misleadingly high detection rates. Consequently, future sanitizer research should focus on implementing more complete sanitizing concepts and conducting thorough functional evaluations to ensure that sanitizers fully realize their conceptual potential.

6. Limitations

Our methodology is designed to measure the memory bug detection capabilities of sanitizers. However, MSET cannot evaluate other functional metrics that lie outside the scope of our assessment. We have identified six limitations, some of which could be addressed by adding specific test case variants to MSET.

False Negatives. The primary objective of MSET is to establish an upper bound on the detection capabilities of sanitizers. It does not account for sanitizer limitations related to metadata handling across complex data flows or in multithreaded scenarios. To ensure compatibility with most sanitizers, MSET's test cases are designed to be simple. If a sanitizer fails such a test, it is likely to fail a more complex version of the same test. The results support this approach, showing that the sanitizers exhibit considerable variability in their capabilities, with most failing to pass all tests, despite their simplicity. However, this may allow potential false negatives caused by conceptual limitations or implementation gaps, e.g., those introduced by race conditions in metadata handling, to escape evaluation.

False Positives. As explained in Section 4.1, MSET uses bug-free versions of the test cases to ensure that a crash in a test case is genuinely due to the included bug. However, a perfect result from MSET does not imply that a sanitizer is free from false positives. Completely ensuring the absence of false positives would require test cases to cover the entire range of valid C/C++ language constructs, which is beyond MSET's scope.

Stdlib Coverage. The MSET test cases utilize `memcpy` and `memset` for reading and writing. Other stdlib functions that access memory are not considered, leaving the stdlib coverage of the sanitizers not fully measured.

Randomization-based Sanitizers. The evaluation of sanitizers that rely on randomization or information hiding, such as Scudo [11] and FreeGuard [12], is conceptually limited. The test cases in MSET always know the memory locations of targets, which undermines techniques like randomizing the locations of objects. Randomly placing guard pages between objects may cause the test cases to fail or succeed unpredictably. To mitigate this limitation, we evaluate each such sanitizer 10 times.

Custom Allocators. The evaluation of sanitizers employing custom allocators yields less informative results. For double-free and misuse-of-free bugs, MSET attempts

to deceive the allocator into returning either free or used memory and determines the type of memory received by knowing which memory location should be erroneously returned by the allocator. However, since MSET's test cases for double-free and misuse-of-free are tailored for glibc, they do not necessarily disrupt custom allocators. When applied to sanitizers with custom allocators, in our evaluation Lowfat [27], [28], [29], EffectiveSan [43], RedFat [32], and FreeGuard [12], these test cases typically result in the custom allocator returning an address to the heap that MSET did not expect. Therefore, for such sanitizers, MSET cannot determine whether it received free or used memory. While this makes the results less informative, it is important to note that they remain *correct*, as the sanitizers permitted the freeing of an invalid pointer.

Other Metrics for Usefulness. MSET is specifically designed to test the bug finding capabilities of sanitizers. Since the overall usefulness of a sanitizer is influenced by several other factors, such as performance, compatibility, and hardware requirements, the evaluation results from MSET should not be interpreted as a general judgment of the sanitizers' overall usefulness.

7. Related Work

The CWE database [64] is a widely recognized resource for security evaluations, offering a comprehensive classification of software and hardware weaknesses. However, it is not intended to serve as a tool for conducting functional security evaluations. As highlighted in Section 3, CWE categories lack the precision necessary for an in-depth evaluation of memory sanitizers, necessitating the creation of a new categorization for MSET.

The Juliet Test Suite [57], developed by the NSA Center for Assured Software, utilizes the CWE categorization to create test cases for the functional evaluation of static code analysis tools. Its test cases typically contain a programming error, the *source* of the error, and a *sink* where the error manifests and the memory violation occurs. Since Juliet is intended to assess the capability of static analyzers to correctly interpret program code, its test cases are duplicated to provide various control-flow and data-flow variants, work with different data types, and cover different sources while maintaining the same sink. For the evaluation of memory sanitizers, this variety of test cases offers no advantages; it is irrelevant how the programming error manifests as long as the memory violation occurs. On the contrary, having numerous test cases that exhibit the same memory violation at runtime dilutes evaluation results: (1) it creates a misleading impression of a sanitizer's security, as a high number of detected test cases might be reported while, in reality, a significant portion are identical at runtime; (2) it reduces comparability, as evaluations use varying subsets of the test cases for different reasons. For the 11 CWE categories relevant to memory safety (out of the 118 categories contained in Juliet), Juliet provides 21,174 test cases, of which 20,674 are variants of 500 unique test cases, with only 199 presenting a unique source/sink combination. In contrast, MSET is specifically designed for testing sanitizers, significantly reducing redundant test cases with identical memory bugs and enhancing the diversity of actual memory errors for a more nuanced evaluation.

Evaluation tools such as BugBench [55] and the one by Zitser et al. [61] offer collections of programs with known bugs, which have practical value but lack the detail necessary for a comprehensive evaluation of memory sanitizers. For example, while SoftBound achieves a 100% detection rate in both tools, MSET reveals that SoftBound's implementation is incomplete, failing to detect intra-object OOBAs in practice. A similar benchmark is UAFBench [59], which aims to evaluate the capabilities of sanitizers to detect use-after-frees and double-frees in a collection of buggy, real-world programs. It is limited in scope, targeting only a subset of memory bugs that are not categorized in relevant ways, and its focus is evaluating compatibility with fuzzing.

The evaluation tool developed by Wilander et al. [60] and its successor, RIPE [58], are purpose-built for the functional evaluation of CFI sanitizers [63], [62]. Similar to MSET, Wilander et al. define their test cases based on multiple dimensions. However, as their tests are designed for CFI, they lack the majority of test cases required for a detailed evaluation of memory sanitizers. Wilander et al.'s original tool [60] provides a limited set of 20 predefined test cases aimed at control flow hijacking. RIPE does not generate code for its test cases but is designed as a self-exploiting program, making the tool more challenging to maintain and the evaluation of sanitizers more complex. MSET and RIPE are specifically designed to test sanitizers that often exist only as proof-of-concept implementations. Using such sanitizers on an entire evaluation tool such as RIPE can be challenging and more error-prone than compiling small test cases. Compiling an entire tool also hinders identifying the cause of a successful or crashing test case. With RIPE, the tool's code surrounding the test cases may always be the cause of a crash. Furthermore, as RIPE is compiled with the sanitizer under test, there are no guarantees that the sanitizer's instrumentation does not unintentionally compromise the integrity of the tool.

X-Ripe [66] and RecIPE [67], successors to RIPE, are designed for enhanced evaluation capabilities and sanitizer compatibility. While X-RIPE remains a self-exploiting tool like RIPE, RecIPE generates code for its test cases similar to MSET. Like RIPE, both tools target CFI sanitizers and lack the test cases required for a detailed evaluation of memory sanitizers. Their test cases always involve a spatial memory bug corrupting a code pointer with the goal of hijacking the control flow to spawn a shell process. The focus of RIPE, X-RIPE, and RecIPE lies in the types of injected shellcode and other methods of corrupting the control flow, such as return-into-libc and return-oriented programming.

Memory Sanitizer Benchmark [68] is a functional evaluation tool with a scope similar to MSET's. However, while it offers 35 static test cases containing memory bugs, it fails to encompass several relevant combinations. For example, when used with EffectiveSan [43], it reports a 100% detection rate for spatial memory bugs.

Yun et al. [69], [70] adopt a different approach for the functional evaluation of hardened allocators, utilizing fuzzing to generate test cases randomly. This approach proves useful in overcoming the randomness often employed by hardened allocators, generating test cases that specifically target their internal details. However, it is not suitable for evaluating general-purpose memory sanitizers.

8. Conclusion

In this paper, we systematically categorized spatial and temporal memory bugs in C/C++ and introduced MSET, our Memory Sanitizer Evaluation Tool, which generates a versatile set of test cases that thoroughly covers these bugs. Our evaluation involved a quantitative comparison of various sanitization techniques and the 16 most relevant memory safety sanitizers. The findings highlight significant variations in the theoretical detection capabilities of these techniques and reveal that, in practice, the implementations of most available sanitizers fall short of their conceptual potential. Furthermore, the evaluation demonstrates the complexities and diversity of memory bugs in C/C++, as well as the challenges associated with detecting them. For instance, our results show that SoftBound+CETS, a conceptually complete sanitizer, misses nearly a quarter of spatial memory bugs in its original implementation, while ASan, likely the most widely used memory sanitizer, cannot detect 50% of use-after-$*$ bugs and any non-linear overflows and underflows. Ultimately, our evaluation concludes that no sanitizer currently provides complete temporal or spatial memory safety.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their valuable feedback, which has improved this work. This research was supported by the German Federal Ministry of Education and Research (BMBF) under Grant No. 16KIS1955 and the Fraunhofer Internal Programs under Grant No. PREPARE 840 231.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *S&P*. IEEE, 2013.

[2] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," in *BlueHat IL*. Microsoft Security Response Center, 2019. [Online]. Available: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/

[3] The Chromium Developers. Memory safety. Google. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[4] The White House, "Back to the building blocks: A path toward secure and measurable software," 2024. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[5] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *ICSE*. ACM, 2018.

[6] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, 2018.

[7] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *S&P*. IEEE, 2019.

[8] B. Perens, "Electric Fence malloc debugger," 1993. [Online]. Available: https://manpages.debian.org/unstable/electric-fence/libefence.3.en.html

[9] Microsoft Corp., "GFlags and PageHeap," 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap

[10] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *DSN*. IEEE, 2006.

[11] LLVM Developer Group, "Scudo hardened allocator," 2023. [Online]. Available: https://llvm.org/docs/ScudoHardenedAllocator.html

[12] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "FreeGuard: A faster secure heap allocator," in *CCS*. ACM, 2017.

[13] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *USENIX Security*. USENIX Association, 2017.

[14] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *USENIX Winter*. USENIX Association, 1992.

[15] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*. ACM, 2007.

[16] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *CGO*. ACM, 2011.

[17] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *CGO*. ACM, 2012.

[18] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX ATC*. USENIX Association, 2012.

[19] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *SecDev*. IEEE, 2020.

[20] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *S&P*. IEEE, 2020.

[21] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating Address Sanitizer," in *USENIX Security*. USENIX Association, 2022.

[22] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *AADEBUG*. Linköping University Electronic Press, 1997.

[23] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *NDSS*. Internet Society, 2004.

[24] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *ICSE*. ACM, 2006.

[25] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *USENIX Security*. USENIX Association, 2009.

[26] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "PAriCheck: An efficient pointer arithmetic checker for C programs," in *ASIACCS*. ACM, 2010.

[27] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *CC*. ACM, 2016.

[28] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *NDSS*. Internet Society, 2017.

[29] G. J. Duck and R. H. Yap, "An extended low fat allocator API and applications," arXiv:1804.04812, 2018.

[30] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CUP: Comprehensive user-space protection for C/C++," in *ASIACCS*. ACM, 2018.

[31] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," Google LLC, arXiv:1802.09517, 2018.

[32] G. J. Duck, Y. Zhang, and R. H. Yap, "Hardening binaries against more memory errors," in *EuroSys*. ACM, 2022.

[33] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "PACMem: Enforcing spatial and temporal memory safety via ARM Pointer Authentication," in *CCS*. ACM, 2022.

[34] K. Hohentanner, P. Zieris, and J. Horsch, "CryptSan: Leveraging ARM Pointer Authentication for memory safety in C/C++," in *SAC*. ACM, 2023.

[35] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, and C. Zhang, "MTSan: A feasible and practical memory sanitizer for fuzzing COTS binaries," in *USENIX Security*. USENIX Association, 2023.

[36] Z. Lin, Z. Yu, Z. Guo, S. Campanoni, P. Dinda, and X. Xing, "CAMP: Compiler and allocator-based heap memory protection," in *USENIX Security*. USENIX Association, 2024.

[37] J. L. Steffen, "Adding run-time checking to the portable C compiler," *Software: Practice and Experience*, vol. 22, no. 4, 1992.

[38] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C programs," *Software: Practics and Experience*, vol. 27, no. 1, 1997.

[39] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *FSE*. ACM, 2004.

[40] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for C," in *PLDI*. ACM, 2009.

[41] Intel Corp., "Pointer checker," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/pointer-checker.html

[42] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *EuroSys*. ACM, 2018.

[43] G. J. Duck and R. H. C. Yap, "EffectiveSan: Type and memory error detection using dynamically typed C/C++," in *PLDI*. ACM, 2018.

[44] B. Orthen, O. Braunsdorf, P. Zieris, and J. Horsch, "SoftBound+CETS revisited: More than a decade later," in *EuroSec*. ACM, 2024.

[45] B. Wickman, H. Hu, I. Yun, D. Jang, J. Lim, S. Kashyap, and T. Kim, "Preventing use-after-free attacks with fast forward allocation," in *USENIX Security*. USENIX Association, 2021.

[46] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "Dangzero: Efficient use-after-free detection via direct page table access," in *CCS*. ACM, 2022.

[47] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, "PUMM: Preventing use-after-free using execution unit partitioning," in *USENIX Security*. USENIX Association, 2023.

[48] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *ISSTA*. ACM, 2012.

[49] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," in *NDSS*. Internet Society, 2015.

[50] Y. Younan, "FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers," in *NDSS*. Internet Society, 2015.

[51] E. van der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable use-after-free detection," in *EuroSys*. ACM, 2017.

[52] S. Ainsworth and T. M. Jones, "MarkUs: Drop-in use-after-free prevention for low-level languages," in *S&P*. IEEE, 2020.

[53] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *ISMM*. ACM, 2010.

[54] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal memory safety via robust points-to authentication," in *USENIX Security*. USENIX Association, 2021.

[55] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, 2005.

[56] MITRE Corp. (2024) About the CVE program. [Online]. Available: https://www.cve.org/About/Overview

[57] NSA Center for Assured Software, "Juliet C/C++ 1.3," 2017. [Online]. Available: https://samate.nist.gov/SARD/test-suites/112

[58] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *ACSAC*. ACM, 2011.

[59] M.-D. Nguyen. (2020) UAF fuzzing benchmark. [Online]. Available: https://github.com/strongcourage/uafbench

[60] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention." in *NDSS*, vol. 3, 2003, pp. 149–162.

[61] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *FSE*. ACM, 2004.

[62] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *S&P*. IEEE, 2019.

[63] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys*, vol. 50, no. 1, 2017.

[64] MITRE Corp. (2023) 2023 CWE top 25 most dangerous software weaknesses. [Online]. Available: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[65] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *CCS*. ACM, 2013.

[66] G. Serra, S. Di Leonardi, and A. Biondi, "X-RIPE: A modern, cross-platform runtime intrusion prevention evaluator," in *OSPERT*, 2022.

[67] Y. Jiang, R. H. Yap, Z. Liang, and H. Rosier, "RecIPE: Revisiting the evaluation of memory error defenses," in *ASIACCS*. ACM, 2022.

[68] MediaKind, "Memory sanitizer benchmark," 2022. [Online]. Available: https://github.com/mediakind-video/memory-sanitizer-benchmark

[69] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *USENIX Security*. USENIX Association, 2020.

[70] I. Yun, W. Song, S. Min, and T. Kim, "HardsHeap: A universal and extensible framework for evaluating secure allocators," in *CCS*. ACM, 2021.

Appendix A. Sanitizer Versions and Selection

The evaluated sanitizers are implemented for various compiler versions and, in some cases, are designed for specific hardware. While most function seamlessly on a Debian GNU/Linux 12 system, some require older Linux versions, such as Ubuntu 16.04. HWASAN [31] is designed for ARM and utilizes the Top Byte Ignore hardware feature, so we ran it on an Ubuntu 22.04 AArch64 machine. The versions of the sanitizers used are listed in Table 2, along with the systems on which they were tested.

TABLE 2. VERSIONS USED FOR SANITIZER EVALUATION

Sanitizer	Source	Version/commit	OS
Baseline	www.github.com/llvm/llvm-project.git	f59e1bc	Debian GNU/Linux 12
ASan [18]	www.github.com/llvm/llvm-project.git	f59e1bc	Debian GNU/Linux 12
ASan-- [21]	www.github.com/junxzm1990/ASAN--	2a954d7	Ubuntu 18.04
Delta Pointers [42]	www.github.com/vusec/deltapointers	5f22ff6	Ubuntu 16.04
Dr. Memory [16]	www.drmemory.org	2.5.0	Debian GNU/Linux 12
EffectiveSan [43]	www.github.com/GJDuck/EffectiveSan	0.1.1-alpha	Ubuntu 16.04
Electric Fence [8]	apt	2.2.5	Debian GNU/Linux 12
FreeGuard [12]	www.github.com/UTSASRG/FreeGuard	bfdf6d9	Debian GNU/Linux 12
HWASAN [31]	apt (clang)	Clang 14.0.0	Ubuntu 22.04 (aarch64)
LowFat [27], [28]	www.github.com/GJDuck/LowFat	20f8075	Ubuntu 16.04
Memcheck [15]	apt	Valgrind-3.16.1	Debian GNU/Linux 12
QASan [19]	www.github.com/andreafioraldi/qasan	113cf03	Debian GNU/Linux 12
RedFat [32]	www.github.com/GJDuck/RedFat	0.3.0 (prebuilt)	Debian GNU/Linux 12
Scudo [11]	www.github.com/llvm/llvm-project.git	f59e1bc	Debian GNU/Linux 12
Softbound+CETS [40], [53]	www.github.com/santoshn/softboundcets-34	9a9c09f	Ubuntu 16.04
Softbound+CETS (rev.) [44]	www.github.com/Fraunhofer-AISEC/softboundcets	7e96d73	Debian GNU/Linux 12

TABLE 3. NUMBER OF TEST CASES PREVENTED BY EACH SANITIZER, SORTED BY BUG TYPE

Sanitizer	Spatial Bug Types			Temporal Bug Types		
	Linear OOBA	Non-linear OOBA	Type confusion OOBA	Use-after-*	Double-free	Misuse-of-free
Total number of test cases	90	72	30	16	4	20
Baseline	36	18	12	0	0	0
ASan [18]	70	18	18	8	4	20
ASan-- [21]	70	18	18	8	4	20
Delta Pointers [42]	60	26	18	0	0	0
Dr. Memory [16]	44	18	14	4	4	20
EffectiveSan [43]	80	72	28	1	4	12
Electric Fence [8]	46	18	16	4	4	20
FreeGuard* [12]	40.7 (σ = 1.5)	17.3 (σ = 1.4)	12.6 (σ = 0.7)	0 (σ = 0)	4 (σ = 0)	8 (σ = 0)
HWASAN* [31]	71.8 (σ = 0.4)	53.9 (σ = 0.3)	24 (σ = 0)	16 (σ = 0)	4 (σ = 0)	12 (σ = 0)
LowFat [27], [28]	60	54	18	0	0	12
Memcheck [15]	48	18	16	4	4	20
QASan [19]	47	16	14	4	4	20
RedFat [32]	48	28	16	4	4	0
Scudo* [11]	36.7 (σ = 1)	19.2 (σ = 0.9)	12 (σ = 0)	0 (σ = 0)	4 (σ = 0)	20 (σ = 0)
Softbound+CETS [40], [53]	72	54	24	16	4	20
Softbound+CETS (rev.) [44]	84	66	28	16	4	20

*Results averaged over 10 runs.

Some sanitizers have implementations that either do not work or are incompatible with MSET when following the authors' instructions. We attempted to run CAMP [36] on Ubuntu 22.04 with Clang 12.0.1 (as instructed), but the compilation failed due to errors. We tried to run RetroWrite [20] on Ubuntu 18.04 and Debian 12 (operating system requirements not specified by the authors); however, running the sanitizer with MSET produced warnings about failed relocations and the inability to find valid sections. Since no errors were reported for the test files provided by the authors, we consider RetroWrite incompatible with MSET. We attempted to run FreeSentry [50] as instructed by the authors, but it failed to compile MSET's test cases.

Some sanitizers with available implementations were excluded because they require uncommon runtime environments. This includes sanitizers designed for very specific hardware features, namely CryptSan [34] and PTAuth [54], which require the ARM PA extension, and LBC [17], which is only available for 32-bit systems. Similarly, DangZero [46] requires kernel modifications that are outside the scope of MSET.

Finally, we excluded sanitizers that rely on one-time allocations, namely FFmalloc [45], PUMM [47], and MarkUs [52], as they mitigate bugs rather than detect them. For these sanitizers, MSET's use-after-$*$ test cases will continuously attempt to reallocate memory, but they will never succeed since memory is never reused. Consequently, the test cases will eventually exhaust the available memory and crash, leaving MSET unaware of what has occurred.

Appendix B. Evaluation Result Numbers

In Table 3, we present the number of test cases per bug type that each sanitizer successfully detected during our evaluation. These numbers were used to calculate the percentages discussed in the evaluation results in Section 5.2 and shown in Figure 5. Scudo [11], HWASAN [31], and FreeGuard [12], rely on randomization. To enhance the accuracy of their results, we calculate the arithmetic mean from 10 runs and provide the standard deviation as well.

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper proposes a systematic approach to test the effectiveness of "memory sanitizers", i.e., tools that dynamically check C or C++ programs for invalid memory accesses. The paper observes that many sanitizers in existing literature are incomplete. In particular, there is no sufficiently complete benchmark that is well-suited for testing sanitizers. Based on a categorization of memory safety bugs, the paper proposes a systematic approach to generating test cases. The resulting benchmark is used to evaluate 16 existing sanitizers, with interesting results.

C.2. Scientific Contributions

- Provides a New Data Set For Public Use
- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science

C.3. Reasons for Acceptance

1) The PC found this work to contribute a valuable benchmark set that the authors have committed to making available for future work. The PC agreed with the authors that existing benchmarks are insufficient.
2) The PC found this work to be effective at showcasing limitations in some sanitizers that are only a consequence of an implementation gap, rather than a theoretical one.

C.4. Noteworthy Concerns

1) Beyond the definition of the primitives used to generate test cases, a significant amount of the remaining work was not as novel (though undoubtedly useful and well-executed).
2) The PC felt that the distinction of this paper's focus on *dynamic analysis* as compared to much prior work's focus on *static analysis* was not as prominent as expected in the paper.