Highly Comprehensive and Efficient Memory Safety Enforcement with Pointer Tagging

Xiaolei Wang, Bin Zhang(✉), Chaojing Tang
National University of Defense Technology
Changsha, China
xiaoleiwang, b.zhang, cjtang@nudt.edu.cn

Long Zhang(✉)
Institute of Systems Engineering, AMS
Beijing, China
zhanglong10@nudt.edu.cn

***Abstract*—Memory safety issues greatly affect the dependability of all software systems. As such, many sanitizers have been proposed to enhance memory safety at the development or deployment phase. However, existing solutions still face many practical challenges, including significant performance overhead, low compatibility with legacy code, and non-comprehensive protection for various program objects and sub-objects.**

In this paper, we propose CECSan, an integrated highly *C*ompatible, *E*fficient and *C*omprehensive compiler-based *San*itizer to detect spatial and temporal memory errors in C/C++ programs. CECSan enforces full memory safety using a integration of pointer tagging, compact metadata table, compiler instrumentation and optimization. The pointer tagging encodes an object metadata index within pointer itself, while instrumentation adds metadata retrival and safety checks for all memory accesses. Specifically, CECSan has advantages over previous sanitizers by simultaneously: 1) supporting memory safety guarantee on sub-object granularity; 2) significantly reducing performance overhead with various optimization strategies; 3) removing the need of a customized memory allocator and avoiding object memory layout changes to improve compatibility. Evaluations on our prototype show that, CECSan outperforms known sanitizers by achieving a bug detection rate of 100% on the Juliet Test Suite. Additionally, CECSan significantly reduces memory consumption while maintaining an acceptable level of runtime overhead.

I. Introduction

Memory safety vulnerabilities are prevalent in programs written in unsafe programming languages such as C and C++. These vulnerabilities are often the root cause of a significant number of security breaches and software dependability. A recent analysis conducted by Google Project Zero [3] revealed that 67% of exploited vulnerabilities in real-world scenarios were attributed to a lack of memory safety. This highlights the critical importance of addressing memory safety concerns in software development. In practice, memory safety vulnerabilities are typically caused by two types of violations [4]: **(1)** temporal safety violations, which involve accessing an object outside of its lifetime, such as use-after-free and double free; **(2)** spatial safety violations, which involve accessing an object outside of its bounds, such as buffer underflow and overflow. These memory safety violations have the potential to compromise memory integrity and program states, leading to memory corruption or even control-flow hijacking.

In recent years, there has been significant attention on memory safety and various sanitizers are proposed to help identify memory safety violations during software testing. Sanitizers are dynamic analysis tools that detect memory-related bugs at runtime by normally incorporating compiler instrumentation. These sanitizers can be broadly categorized into two groups: ❶ Location-based (redzones) methods [5], [9], [21]–[23], [26] aim to identify memory accesses to invalid memory regions. Notable examples include AddressSanitizer (ASan) [5]. ❷ Identity-based (metadata) methods aim to detect memory accesses that do not target their intended referent. They can be further divided into **Per-object based** based [6], [8], [10], [11], [16], [17], [20], [34] and **Per-pointer based** [7], [12], [13], [15], [18], [19], [24], [25], [29] approaches. While existing works have demonstrated excellent improvements in memory safety from various perspectives, there are still several limitations and challenges that may hinder practical adoption, as follows:

① **Partial memory safety guarantees.** In practice, many sanitizers prioritize performance over precision, leading to a compromise in accuracy with the acceptance of false negatives. For example, SoftBound [6] and LowFat [34] primarily focus on providing spatial memory safety, while CETS [28] and the recent UAFSan [19] concentrate solely on temporal memory safety. Notably, UAFSan exclusively targets heap objects, disregarding the stack and global variables. Even widely deployed sanitizers such as ASan [5] and the more recently proposed sanitizer PACMem [12] fall short in detecting sub-object overflows, primarily due to performance considerations. Consequently, these tools are often more adept at aiding in troubleshooting than ensuring robust security measures.

② **Significant performance overhead bottleneck.** Most sanitizers designed to ensure comprehensive memory safety assurances typically come with substantial performance overheads. To illustrate, ASan [5] incurs a 2.5x CPU runtime overhead and a 3.37x memory overhead on average. Although HWASan [29] leverages hardware features to mitigate memory overhead, its runtime overhead remains around 2x. The notable performance burdens associated with existing sanitizers impede their efficiency, significantly limiting their applicability.

③ **Limited compatibility with uninstrumented or legacy code.** Many existing sanitizers sacrifice compatibility for the sake of precision and performance. This trade-off often disrupts the default assumptions of programmers and may lead to compatibility issues, particularly when linking with uninstrumented code. For instance, ASan adopts a custom heap allocator and replaces the default allocator that still utilized by uninstrumented code. This substitution may alter the functionality of the protected software. Moreover, the introduction of fat pointers [8] alters the memory layout, presenting compatibility challenges with external modules, particularly precompiled libraries.

In summary, none of the existing sanitizers manages to achieve full memory safety without compromising compatibility and efficiency. To this end, we introduce CECSan, a novel highly **C**ompatible, **E**fficient and **C**omprehensive **S**anitizer designed to ensure full memory safety for all C/C++ objects on 64-bit architectures. CECSan is a software-only tool comprising a compiler and a runtime library. To improve performance, CECSan utilize the currently unused top 16 bits of a 64-bit pointer (typically only 47 bits of pointers are used on x86-64, and 48 on ARM64) to associate metadata information with each pointer, ensuring implicit propagation regardless of the pointer's use. In this way, the overhead associated with metadata propagation is minimized. To optimize memory usage, CECSan records base and bound metadata information in a compact and reusable table. This table is flexibly indexed and accessed using pointer's unused most-significant bits.

Unlike most existing sanitizers, to maintain compatibility with legacy code, CECSan conducts compile-time instrumentation during Link Time Optimization (LTO), which could identify external calls comprehensively. Moreover, CECSan provides full memory safety guarantee for all program objects (globals, stack, heaps). It achieves this by precisely tracking an objects's lifetime and performing memory safety checks at proper locations. It is worth noting that CECSan does not rely on a custom memory allocator but rather adopts fine-grained code instrumentation coupled with various compiler optimizations. This approach not only helps improve compatibility but is also applicable in scenarios requiring an unaltered memory layout, such as vulnerability exploitability analysis, etc. Note that other types of bugs (e.g.., uninitialized variables) except memory safety related, are out of the scope of this paper.

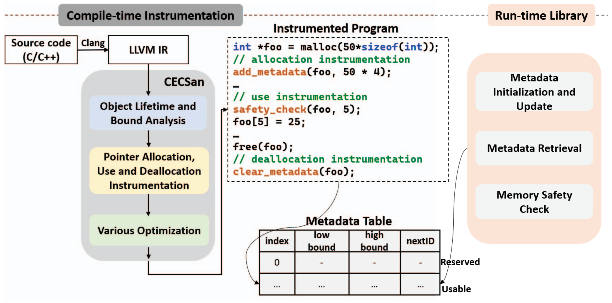
The main contributions are summarized as follows:

- We design a new reusable and compact metadata table and sub-object overflow detection method to achieve memory safety at both object and sub-object granularity.
- We engineer the integration of various technologies and optimization strategies in pursuit for compatibility and performance.
- We design a new automation framework to evaluate CECSan systematically in terms of security. Evaluation demonstrates that CECSan exhibits the capability to detect multiple classes of memory safety violations.
- A full LLVM-based prototype of CECSan[1] is implemented to enhance full memory safety for C/C++ applications on x86-64 and ARM64 architectures.

II. CECSan Design

In this section, we describe the design of CECSan. We begin by providing an overview. Subsequently, we detail the metadata management for pointers, and comprehensively describe the memory safety enforcement for both objects and sub-objects. Finally, we show how to improve the compatibility and implement performance optimizations.

[1]https://github.com/040840308/CECSan.git



```
int *foo = malloc(50*sizeof(int));
// allocation instrumentation
add_metadata(foo, 50 * 4);
…
// use instrumentation
safety_check(foo, 5);
foo[5] = 25;
…
free(foo);
// deallocation instrumentation
clear_metadata(foo);
```

index	low bound	high bound	nextID
0	-	-	-
...	...	...	...

Fig. 1: Overview of CECSan

A. *System Overview*

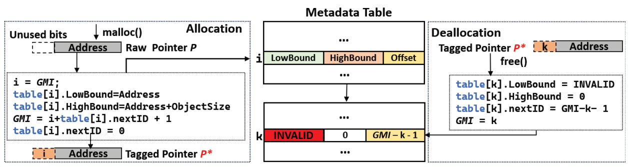
Following common practice, CECSan tracks objects' metadata, conducting metadata checks before any memory accesses. Typically, metadata is *generated* during object allocation, *propagated* during pointer operations, *checked* when pointers are dereferenced, and *cleaned up* upon object deallocation. Each of these steps involves particularly time-consuming metadata propagation and checking. CECSan strategically leverages the unused high bits of pointers to eliminate metadata propagation and accelerate metadata checking.

As illustrated in Figure 1, CECSan consists of compiler-based instrumentation and runtime library. At compile time, CECSan analyzes the bound and liveness properties of objects. It strategically inserts instrumentation at appropriate locations to either track metadata or conduct memory safety checks. Specifically, when an object is *allocated*, CECSan generates metadata, which is placed in an indexed table and embeds the corresponding index into pointers associated with this object(i.e., `foo` in Figure 1). The index will implicitly propagate along with pointers, thereby saving overhead of metadata propagation. Furthermore, before a pointer is *dereferenced*, the index is used to lookup the metadata table, enabling efficient metadata checks. Finally, when an object is *deallocated* through pointers, its validity will be verified. Upon successful verification, the object's metadata is cleaned.

B. *Metadata Management*

To check memory access safety at run-time, CECSan keeps metadata in a disjoint table and uses pointer tagging for linking pointers to an object's metadata. Next, we first introduce the CECSan metadata structure. Then, we discuss the metadata management throughout the C/C++ object's lifespan, which generally includes *Allocation*, *Access*, *Deallocation*.

1) Metadata Structure: As shown in Figure 1, CECSan stores all metadata disjointly from pointers and objects in the metadata table, presented as a linear array for performance considerations. Each table entry uniquely corresponds to one object and contains metadata represented as follows: $(low\ bound, high\ bound, nextID)$. The fields *(low bound,*



```
i = GMI;
table[i].LowBound=Address
table[i].HighBound=Address+ObjectSize
GMI = i+table[i].nextID + 1
table[i].nextID = 0
```

```
table[k].LowBound = INVALID
table[k].HighBound = 0
table[k].nextID = GMI-k- 1
GMI = k
```

Fig. 2: Metadata Management for Object Allocation and Deallocation

high bound) are used for spatial memory safety check. Notably, the *low bound* is also repurposed to enforce temporal memory safety (**§II.C**), while *nextID* is employed to implement a free list encoded in the table, reusing metadata entries as early as possible (described later). To efficiently manage metadata, CECSan maintains a global variable GMI to indicate the current metadata table index. GMI is initialized with value 1 and is incremented after each new allocation. Note that the operations on GMI are arranged in a thread-safe way.

2) Metadata Creation: As shown in Figure 2, when allocating an object, a new metadata entry is generated at the current table index GMI. The new entry records the object's base address as the *lower bound* and the result from adding the object's size to its base address as the *upper bound*. The *nextID* of object is also set to 0 and reset once the object is deallocated. The GMI is also be incremented based on the current index I and *nextID* value to achieve a free list. After adding a new table entry, by embedding the entry index i into pointer's high-order bits, the raw pointer is tagged and returned as the definitive pointer to the object.

3) Metadata Tracking and Access: As stated above, once the metadata is created for an object, the embedded index will propagate along with the tagged pointer automatically, which implicitly propagates metadata from source to destination pointers through pointer derivation, arithmetic or assignment. For a tagged pointer, we can extract the index from its high unused bits and subsequently retrieve objects' metadata. In general, only a 47-bit address space is used for x86-64 user-space programs. Consequently, CECSan sets the metadata index size to 17 in the x86-64 environment.

4) Metadata Cleanup: Before the object's pointer can be used for deallocation, its spatial and temporal integrity is verified, preventing deallocation through an illegal or invalid pointer. For instance, if the object to be freed is a heap object, the pointer for deallocation should represent the base address of a valid object. This ensures that the pointer is not a dangling pointer and does not point inside the object. Therefore, an additional check is performed before clearing the metadata table of the heap object. After successful verification, the object metadata entry is overwritten by setting *low bound* with *INVALID* (a very high value), *high bound* with zero. Specifically, the field *nextID* is set to the offset to the next available table entry ($GMI - k - 1$) and the currently released index is set as k to be used next time, thus maintaining the free list correctly. C code illustrating these operations is shown in Figure 2. Finally, the raw pointer is used to free the object's memory through the standard deallocation.

Algorithm 1 Optimized Pointer Deference Check.

```
1: procedure DEREFERENCE_CHECK(POINTER, ACCESS_SIZE)
2:     $tag|address \leftarrow pointer$ // strip tagged pointer
3:     $lbound \leftarrow Table[tag].low\_bound$ //Table: Metadata Table
4:     $hbound \leftarrow Table[tag].high\_bound$
5:     $check \leftarrow (address - lbound)|(hbound - address - access\_size)$
6:     $check \leftarrow check \ \& \ 0x8000000000000000$
7:     if $check! = 0$ then
8:         $UAF \ or \ OOB \ Detected!$
9:     $return$
```

C. Enforce Memory Safety

CECSan provides memory safety guarantees for heap, stack, and global objects. By examining all possible pointer operations, we observe that only pointer dereference and deallocation are safety related. Thus, the metadata check is performed only when pointer is **dereferenced** or **deallocated**. In this way, critical memory safety checks are applied precisely, minimizing unnecessary overhead for other pointer operations.

1) Pointer Dereference Checking: In this phase, each pointer dereference triggers a memory safety check to ensure that the dereference must be confined within the bounds of the object and during the object's allocated lifespan. CECSan employs bound checks to enforce spatial memory safety. Specifically, it utilizes the pointer's high unused bits as an index to retrieve the intended object's metadata, which are then used to verify the legality of the access.

It is crucial to note that the bound check implemented in CECSan also inherently provides temporal safety guarantees. As mentioned above, when an object is deallocated and its pointer becomes dangling, its metadata entry is overwritten by setting the low bound to an *INVALID* address. In cases where the cleared entry has not been taken by other objects, the retrieved low bound is an *INVALID* address, causing the bound check to fail. Even if the corresponding entry has been claimed by another object, which accidentally has the same index, the retrieved low and high bound are unlikely to match those of the freed object. Consequently, the bound check will still fail. As a result, this bound check mechanism is sufficient to catch temporal safety bugs, including use-after-free bugs.

To improve performance, we implement an optimized check that combines the temporal and spatial checks, as illustrated in Algorithm 1. **On one hand**, in the absence of spatial memory errors, we observe that the difference between the dereferenced pointer and the low bound should always be greater than or equal to 0 (line 5). Similarly, the high bound minus the pointer should always be greater than or equal to 0 (line 5). Consequently, the high order bit in the differences should always be 0. We OR these two differences together, mask off the low order 63 bits, and then the final result should be 0 (line 6). Thus, a single memory safety check can be executed to detect Out Of Bound (OOB) errors (line 7). **On the other hand**, when accessing a deallocated object, since the *low bound* of its metadata is set with high invalid address, causing the same safety check to fail. In this way, the temporal safety guarantees (e.g.,UAF) are also provided. Once the safety check succeeds, pointers will be stripped (i.e., their tag bits are removed) and dereferenced.

Algorithm 2 Pointer Deallocation Check.

```
1: procedure DEALLOCATION_CHECK(POINTER, ACCESS_SIZE)
2:     $tag|address \leftarrow pointer$ // strip tagged pointer
3:     $lowbound \leftarrow Table[tag].low\_bound$ //Table: Metadata Table
4:     $check \leftarrow (address - lowbound)$
5:     if $tag! = 0$ && $check! = 0$ then
6:         $Double\ Free\ or\ Invalid\ Free\ Detected!$
7:     $return$
```

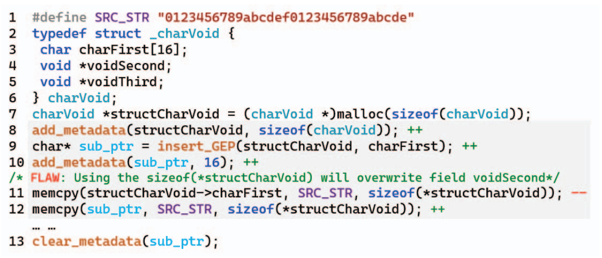
2) Pointer Deallocation Checking: Similarly, the pointer's temporal and spatial validity are also verified before deallocation (shown in Algorithm 2), prohibiting illegal free or invalidation. Consider a legal heap pointer, before deallocation, it should be verified to point to a valid object's base address. **First**, we compare the low bound of pointer's metadata with its real address (line 4) to prevent free-inside-buffer errors where free pointers are not an object's base address. **Second**, since we reset the *low bound* of an pointer's metadata as *INVALID* address after legally deallocating it, thus, the safety check at line 4 also implicitly prevents freeing the dangling pointers. Thus, CECSan can also catch double-free and invalid-free bugs. After successful verification, CECSan first invalidates the corresponding metadata entry, as detailed in §II.B. Then the object's memory is freed using the standard deallocation.

3) Protection for Stack and Global Objects: In C/C++, heap objects are typically accessed through pointers. Therefore, CECSan can instrument and protect them straightforwardly as described above. However, most stack objects are local variables that are safely accessed directly through the stack pointer. Thus, CECSan focuses on instrumenting unsafe stack objects, which are mainly *arrays* and *buffers*. The instrumentation involves creating metadata in function *prologues*, clearing the metadata in function *epilogues*, and inserting checks when accessing objects. The distinction between safe and unsafe objects on the stack is based on whether their addresses are taken or their accesses can be statically guaranteed to be in-bounds.

For global objects, CECSan applies the same static analysis as for the stack, only instrumenting unsafe ones that might potentially corrupt other objects. Compared to heap and stack objects, the global object's scope poses an additional challenge. Since global objects are allocated statically, non-PIC executables access them statically as well, requiring no pointers to be used at all. To secure unsafe global objects, CECSan borrows the approach from [36] and employs one Global Pointer Table (GPT), which contains a tagged pointer for each unsafe global object and is accessible as global variable itself. During instrumentation, accesses to the original global objects are modified to first load the corresponding tagged pointer from the GPT and then check the pointer to ensure memory safety. The code for generating and and storing the tagged pointer in the GPT is instrumented at the beginning of the main function of the instrumented program. Note that since all accesses to the GPT are added by CECSan itself, loads and stores to it are not instrumented with checks.

D. Sub-Object Granularity Memory Safety Violation Detection

A sub-object level memory safety violation occurs due to an over- or under-flow between fields of the same object.



```
1  #define SRC_STR "0123456789abcdef0123456789abcde"
2  typedef struct _charVoid {
3   char charFirst[16];
4   void *voidSecond;
5   void *voidThird;
6  } charVoid;
7  charVoid *structCharVoid = (charVoid *)malloc(sizeof(charVoid));
8  add_metadata(structCharVoid, sizeof(charVoid)); ++
9  char* sub_ptr = insert_GEP(structCharVoid, charFirst); ++
10 add_metadata(sub_ptr, 16); ++
/* FLAW: Using the sizeof(*structCharVoid) will overwrite field voidSecond*/
11 memcpy(structCharVoid->charFirst, SRC_STR, sizeof(*structCharVoid)); --
12 memcpy(sub_ptr, SRC_STR, sizeof(*structCharVoid)); ++
   … …
13 clear_metadata(sub_ptr);
```

Fig. 3: Code Example of Sub-Object Overflow

This type of violation is particularly relevant for programs that extensively use structs and classes that can have multiple fields. Figure 3 shows a code snippet that illustrates a sub-object level overflow. In this case, the size calculation $sizeof$ for the memcpy function is based on the entire $struct$ rather than a specific buffer size (16 byte), leading to a memcpy call with a size larger than the target buffer. As a result, a sub-object overflow occurs, corrupting adjacent data fields and potentially leading to hijacking attacks if the fields contain function pointers. Most current sanitizers, such as known ASan [5] and PACMem [12], cannot detect this kind of sub-object memory safety violation. The main reason behind this is because they always treat objects as a single unit without differentiation between sub-objects.

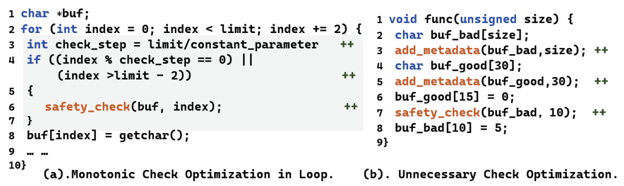
To solve this, CECSan employs two key techniques. **First**, derive sub-object type and calculate bounds. C/C++ types inherently encode bounds information (e.g., int[100]), meaning that type and bounds checking go hand-in-hand. Inspired by this, for each sub-object use, CECSan firstly identify its type and uses type information to derive sub-object bounds. **Second**, narrow sub-object access bounds. As shown in Figure 3, the basic idea is as follows: for a sub-object access (e.g., structCharVoid->charFirst), CECSan creates a temporary sub-object pointer sub_ptr and then track its bounds metadata through instrumentation (line 9, 10). Simultaneously, the original sub-object access is replaced by the newly generated pointer sub_ptr(line 12). This allows to narrow sub-object bounds through metadata lookup and detect the potential sub-object level overflows at runtime. The sub-object metadata is cleared when it goes out of scope (line 13). These techniques enable CECSan to effectively handle sub-object related memory safety issues, providing a more comprehensive memory safety solution. Since arrays may present inside composite types (e.g., struct), CECSan could recursively look through all elements inside a composite type and perform similar instrumentation on demand.

E. Compatibility with External Uninstrumented Code

Since CECSan utilizes upper bits of pointers to store indexes, it may cause compatibility issues if pointers are used across instrumented and uninstrumented code. Therefore, CECSan takes extra steps to deal with such cases. **First**, for those untagged pointers returned from uninstrumented code, CECSan will use a reserved metadata entry (i.e., tag index is 0, minimum base address and maximum upper bound) for it, since it does not know how the object is created. And CECSan is able to use them as-is without breaking any functionality without any safety check. **Second**, when passing tagged pointers as arguments to external functions, the pointers are checked and stripped by CECSan before issuing the function call. For functions that return one of their pointer arguments, CECSan goes a step further by wrapping the call to temporarily store the stripped tag and reapply it to the returned pointer argument after the function returns. To identify functions external to CECSan-instrumented code, CECSan performs instrumentation during Link Time Optimization (LTO), enabling our instrumentation to detect truly external functions that will not be instrumented. Consequently, CECSan achieves complete compatibility with any external, uninstrumented code, particularly in scenarios involving dynamically linked libraries.

F. *Performance Optimizations*

In essence, CECSan requires instrumentation for each memory access to ensure memory safety. However, in certain scenarios, checks can be omitted when the underlying memory access is deemed statically safe or redundant. To minimize the overhead of CECSan while maintaining security guarantees, in addition to existing optimization measures (such as redundant and recurring checks elimination used in ASAN- -'s [23]), we have also implemented the following two optimizations.



```
char *buf;
for (int index = 0; index < limit; index += 2) {
 int check_step = limit/constant_parameter   ++
 if ((index % check_step == 0) ||
      (index >limit - 2))                     ++
 {
    safety_check(buf, index);                 ++
 }
 buf[index] = getchar();
 … …
}
```

(a).Monotonic Check Optimization in Loop.

```
void func(unsigned size) {
 char buf_bad[size];
 add_metadata(buf_bad,size); ++
 char buf_good[30];
 add_metadata(buf_good,30);  ++
 buf_good[15] = 0;
 safety_check(buf_bad, 10);  ++
 buf_bad[10] = 5;
}
```

(b). Unnecessary Check Optimization.

Fig. 4: Examples of Checks Optimization

1) Loop Oriented Check Optimization: To further enhance performance, we concentrate on optimizing checks within loops, specifically focusing on **invariant checks** and **monotonic checks**. Firstly, we identify invariant checks inside loops that operate on a constant pointer and move them out of loops. This optimization eliminates numerous duplicate checks within loops, and a single deduplicated check relocated after the loop, is sufficient. Notably, previous redzone-based approaches [22] limit this optimization to loads and cannot move checks on store operations outside of loops due to the potential overwriting of redzones. In contrast, CECSan, not relying on redzones, can simultaneously handle both loads and stores. Monotonic checks in loop refer to consecutive memory accesses read or write to the same object, but increment or decrement the pointer each time. To identify such checks, we use the LLVM's scalar evolution framework to determine linear access behavior. For instance, in a loop over an array, the scalar-evolution indicates the first element accessed and how the index changes or evolves on each iteration of the loop. If the start and indexing steps are known and bounded, we consider the related checks as monotonic and record the maximum access range, start range, and indexing step. To optimize monotonic checks, we introduce a new constant check method that is independent of the maximum range of the loop. As depicted in Figure 4(a), based on the statically determined max access range $limit$, we generate a new variable $check\_step$ with specified number of constant checks (default parameter is 5). We perform a safety check only when memory access is integer times of $check\_step$ or close to the maximum range. This approach allows us to group multiple checks into one without compromising safety check capability.

2) Type Information based Unnecessary Check Removal: A straightforward approach to guard against out-of-bound access is to include range checks on every pointer arithmetic operation, ensuring that pointers do not exceed memory bounds. An optimization strategy is that if the compiler knows the pointer is in-bound and safe, the validation of which could be removed to improve the performance. However, the compiler typically lacks information about the memory range of a pointer, making it challenging to determine which pointers are within bound and access-safe. To implement the optimization, we use type information to ascertain the memory range of a pointer during the compilation process. **First**, we exclude non-composite objects that are not involved with pointer arithmetic, e.g., int-typed objects, from tracking. It is unnecessary to perform bounds checking for pointers to them. We filter out these easily recognizable, simple cases to avoid unnecessary checks. **Second**, instead of full bounds checks, we ignore pointers involved in safe pointer arithmetic which could be statically proven in-bound with respect to its base object. For example, it is a field access or an array access with constant in-bounds index, as the $buf\_good[15]$ shown in Figure 4(b). Especially, this case only applies when the bounds can be determined statically and the index is smaller than the number of elements. It is worth emphasizing that this optimization is not applied to pointers with no type information (e.g., void*) or when their type sizes cannot be determined at compilation time.

III. Implementation Details

We have implemented a prototype of CECSan for the 64-bit architectures (i.e., x86-64 and ARM64) as an extension to the LLVM compiler framework (version 15.0.0). The CECSan prototype comprises two main components: (i) a custom compiler extension designed for instrumentation and (ii) a runtime support library for managing metadata for program objects. The current prototype supports C and C++ programs.

Compiler Extension. The compile-time instrumentation and optimization are implemented during Link Time Optimization (LTO) phase. Specifically, after combining all the object files into one file, we first identify truly external function calls before the instrumentation. Subsequently, we insert the memory safety checks and inline all our CECSan runtime library functions at proper locations. To tag allocated pointers for metadata tracking, CECSan is able to intercept various allocations via the `libc` allocator, and ensure that the metadata is generated and linked to its corresponding pointers. Notably, the CECSan instrumentation is adept at intercepting GNU C/C++ standard library's allocation and deallocation functions for seamless metadata management. Currently, it supports all common memory allocation APIs such as `malloc`, `free`, `new`, and `delete`.

Runtime Library. The runtime component is responsible for the initialization and maintenance of the metadata table, as well as the handling of memory safety violations. The runtime library adds a constructor function to protected program, which is executed by the dynamic linker at load-time and will run at program startup. Using this constructor, the runtime library mainly allocates and initializes a metadata table through `mmap` before program starts. Each metadata entry corresponds to 24 bytes of metadata, comprising 8 bytes for the *lower bound*, 8 bytes for the *upper bound*, and 8 bytes for the *nextID*. Within the metadata table, the entire table is initialized with value 0x00 for all entry fields. The *upper bound* of reserved metadata entry (i.e., index is 0) is initialized as very high address. Depending on the architecture and its number of unused bits T in a pointer, the metadata table has a constant size of 2^T entries (currently 2^{17} in prototype). As evaluated in [12], [18], several long-running and large real world programs (e.g., Nginx and Apache) have less than 100 thousand live objects, which is far from the metadata table's size. Furthermore, only at most 0.1% and 0.2% of entries were used at the same time. Thus, we believe the metadata table is sufficient for practical use. The constructor also initializes the global metadata table index GMI to 1. The index is also allocated in the runtime library, making it solely accessible to our metadata handling routines. To ensure thread-safety for metadata index, CECSan's runtime library uses a global mutex to provide atomic access to the metadata table index GMI. The metadata table and its index are independent for each process so that there are no inter-process conflicts.

IV. Evaluation

In this section, we evaluate both the security and performance overhead of our CECSan prototype. We first evaluated the memory safety related bug detection capabilities of CECSan, using the public Juliet Test Suite and Linux Flaw project [2]. We draw comparisons with various established sanitizers to highlight the security guarantees offered by CECSan. Subsequently, we evaluate the performance overhead of CECSan. Specifically, we measured runtime and memory consumption overhead using the SPEC CPU 2006 and CPU2017 benchmarking suites. Additionally, for SPEC CPU2017, we enable OpenMP where available for assessment purposes.

A. Environment and Comparison Targets

We evaluated our CECSan prototype on a machine with Ubuntu 20.04, 128 GB RAM, and an Intel Xeon-6254 CPU of which we use the 32 performance cores. During security evaluation, we compared our CECSan prototype with other memory safety solutions, namely unmodified CryptSan [36], HWASan [29] and the sanitizers ASan [5], Softbound/CETS [6], [28]. These solutions represent various approaches, such as pointer tagging, memory tagging, red-zones, and bounds-checking, respectively. During performance evaluation, we make fair comparison with two known sanitizers ASan [5] and ASAN– [23], which are open-source and have been shown to

TABLE I: Description of Juliet Test Suite

CWE Name	Vulnerability Type	Number of Samples
CWE121	Stack Buffer Overflow	4896
CWE122	Heap Buffer Overflow	3777
CWE124	Buffer Underwrite	1440
CWE126	Buffer Overread	2004
CWE127	Buffer Underread	2000
CWE415	Double Free	818
CWE416	Use After Free	393
CWE761	Invalid Free	424
Total	–	15752

TABLE II: Comparison of Memory Violation Detection

Name (#cases)	CECSan (15752)	PACMem (11531)	CryptSan (5364)	HWASan (5364)	ASan (15752)	SoftB/CETS (3970)
CWE121	100%	*98.82%*	*98.5%*	*82.9%*	*83.74%*	**77.7%**
CWE122	100%	*99.01%*	*97.4%*	*94.6%*	*83.92%*	**73.7%**
CWE124	100%	100%	100%	*81.9%*	*80.18%*	**82.5%**
CWE126	100%	100%	100%	*99.7%*	*82.89%*	**96.5%**
CWE127	100%	100%	100%	*75.9%*	*91.01%*	**78.4%**
CWE415	100%	100%	100%	100%	100%	100%
CWE416	100%	100%	100%	*50.9%*	*90.41%*	**51.3%**
CWE761	100%	100%	100%	*0%*	*91.56%*	100%

have good performance. Unfortunately, we could not measure PACMem's [12] runtime overheads since it is not open-source.

B. Security Evaluation

1) Juliet Test Suite: The Juliet Test Suite is a collection of C/C++ test cases covering a variety of CWEs. As shown in Table I, we only consider known test cases of specific CWEs introducing temporal and spatial memory corruptions, including heap overflow, use after free, etc. For each test case there is a good version without any memory violation and a bad version containing memory corruption. Most previous works exclude test cases that depend on external input (e.g., `fgets`) to easily enable automation, thus the limited number of evaluated test cases is used, e.g., 11531 and 5364 for PACMem [12], CryptSan [36], respectively. In contrast, for our evaluation, we design a new automation framework with dummy server to provide external inputs (e.g., `socket`) and does not exclude any test cases, resulting in a total of 15752 test cases listed in Table I. However, for the evaluation of Softbound/CETS, additional cases leading to compilation errors had to be excluded, resulting in only 3970 cases.

Table II shows the results of evaluating CECSan and related sanitizers with Juliet Test Suite, and we can make the following observations: **1**). Most existing sanitizers have much more *false negatives* both in terms of implementation and design, i.e., they would miss real vulnerabilities at runtime. For example, in terms of design, both ASan and HWASan cannot detect sub-object overflow cases. Additionally, HWASan cannot detect invalid free bugs (i.e., CWE761 shown in Table II). While the bounds tracking used in Softbound/CETS should be able to detect more bugs, the released source code lacks features and is missing various wrapper functions. **2**). None of the approaches except SoftBound/CETS have **false positives**. Regarding false positives, there are some flaws in the prototype implementation of SoftBound+CETS, and thus it has a high false positive rate. **3**). Recent PACMem and CryptSan have even fewer false negatives than ASan and HWAsan, but they still have false negatives caused by sub-object overflow, due

TABLE III: Vulnerability Detection on Linux Flaw project.

CVE	Type	Detected by CECSan?
CVE-2006-2362	stack-buffer-overflow	✓
CVE-2007-6015	heap-buffer-overflow	✓
CVE-2009-2285	heap-buffer-overflow	✓
CVE-2013-4243	heap-buffer-overflow	✓
CVE-2014-1912	heap-buffer-overflow	✓
CVE-2015-8668	heap-buffer-overflow	✓
CVE-2015-9101	heap-buffer-overflow	✓
CVE-2016-10095	stack-buffer-overflow	✓
CVE-2017-12858	heap-use-after-free	✓
CVE-2018-9138	stack-overflow	✓

TABLE IV: Performance Overhead Comparison on SPEC2006

	Runtime Overhead			Memory Overhead		
Benchmark	ASan	ASAN–	CECSan	ASan	ASAN–	CECSan
400.perlbench	307%	306%	277%	345%	346.6%	1.94%
403.gcc	147%	146%	220%	182%	187.5%	2.92%
429.mcf	60.5%	65.47%	174.8%	12.68%	12.69%	0.34%
447.dealII	63.4%	66.28%	162.2%	124.3%	124.4%	2.06%
458.sjeng	90.3%	90.81%	127.3%	2.53%	2.53%	3.53%
462.libquantum	14.1%	40.6%	62.39%	265.9%	266%	5.65%
470.lbm	48.08%	60.58%	148.6%	13.31%	13.28%	1.41%
471.omnetpp	144.9%	98.64%	106.8%	342%	342%	3.71%
Average	109.4%	109.3%	189.7%	160.9%	161.9%	2.69%
Geometric Mean	89.46%	76.99%	140.52%	70.67%	71.13%	2.47%

to the trade-off with metadata tracking overhead. Although SoftBound/CETS claims to detect sub-object overflow, it does not detect any of the sub-object overflow test cases in the Juliet test. **4)**. CECSan is the only sanitizer with complete detection of all temporal and spatial bugs on the most numerous examples, and achieves a 100% vulnerability detection rate without false positives and negatives. This is mainly contributed to comprehensive instrumentation and design. For instance, through instrumenting wide character related functions (e.g., `wcsncpy`) that were previously overlooked by most sanitizers, CECSan can detect more heap buffer overflows from CWE122. Furthermore, aided by our sub-object level memory safety violation detection, CECSan is the only sanitizer able to detect all sub-object overflow cases in CWE 121 and 122, such as CWE122*overrun_memcpy_17.c.

2) Linux Flaw Project: In the second experiment, we evaluate CECSan's bug detection capabilities on certain CVEs of the Linux Flaw project [2]. After visiting all the vulnerabilities in the database, we reproduce 10 of them and omit some bugs that were not reproducible on our machine. Details of the vulnerabilities are presented in Table III. CECSan can detect all 10 memory safety related vulnerabilities. This experiment could demonstrate the bug detection capability of CECSan to some extent.

C. Performance Overhead

In this section, we evaluate the performance of CECSan in terms of runtime and memory overhead. We measure performance using the SPEC CPU2006 and CPU 2017 benchmarking suites. To put the performance of CECSan into better perspective, we make comparison with known sanitizers having good performance, namely ASan [5] and ASAN– [23]. We compiled all benchmarks with optimization level O2 and executed them in single-threaded mode.

Runtime Overhead. Table IV and V shows the runtime overhead for CECSan, ASan, and ASAN– on the SPEC2006 and SPEC2017 with x86 architecture. On SPEC2006, the average runtime overheads for ASan and ASAN– are 109.4% and 109.3%, respectively. And the average runtime overhead of CECSan is 189.7%, which is around 73% slower than ASan and ASAN–. When looking at some benchmarks such as `400.perlbench`, CECSan runs faster than the ASan and ASAN–. This can be explained by the faster integrated memory security check as well as the more aggressive compiler optimizations. On SPEC2017, the average runtime overhead of CECSan is 187.5%, which is around 70% slower than ASan. Combining the average and geometric mean results, we can find that CECSan's runtime overhead is indeed higher compared to ASan and ASAN–. However, we want to argue that this is inevitable due to the need of frequent metadata checks for providing stronger security guarantees.

TABLE V: Performance Overhead Comparison on SPEC2017

Performance		Average	Geometric Mean
Runtime Overhead	ASan	110.2%	83.0%
	ASAN–	103.8%	85.5%
	CECSan	187.5%	136.7%
Memory Overhead	ASan	1260.0%	204.3%
	ASAN–	826.5%	172.0%
	CECSan	5.1%	3.9%

Memory Overhead. Table IV and V also show the average memory overhead of the sanitizers on SPEC2006 and SPEC2017, respectively. It is easy to see that both ASan and ASAN– have a very high memory overhead. In contrast, the average memory overhead CECSan introduces on SPEC2006 and SPEC2017 is 2.69% and 5.1%, respectively. Therefore, regarding the memory overhead, CECSan uses notably less memory, which is a logical consequence from the fact that we do not employ shadow memory, but rather compact metadata table.

Summary. In conclusion, the benchmarking experiments on SPEC CPU 2006 and 2017 highlight the benefit of implementing a sanitizer using pointer tagging. Overall, on one hand, CECSan has acceptable runtime overhead compared to ASan. On the other hand, its memory overhead is significantly lower than ASan's. However, CECSan provides stronger memory safety guarantees than ASan. Considering all performance overheads and security detection capabilities, CECSan is a practical choice for balancing performance and security.

V. Discussion

Below we discuss limitations of CECSan and future works.

1. The first limitation of CECSan results from the number of unused bits in a pointer. As those bits store the metadata table index *I*, they directly determine the maximum number of objects that can be represented in the metadata table. To handle cases involving a large number of objects, we design a free list to reuse the deallocated metadata table slots. In this way, the situation of tight metadata data table space can be alleviated to some extent. However, there still may be extreme cases that the metadata table is exhausted. To address this, one option is to use techniques like linked lists for storing conflicted metadata. However, this may introduce high performance overhead and more investigations are needed in future.

2. As in previous work, CECSan cannot handle the corner case of external global arrays defined in uninstrumented code.

CECSan assumes that global objects are allocated statically and easy to recognize. If the global was defined in uninstrumented code, this assumption does not hold. Consequently, CECSan will fail to enhance memory safety for such globals. Note that such a situation is not usual during our experiment, as we mainly support full user-space instrumentation.

3. CECSan allows instrumented programs to link against uninstrumented code (e.g., legacy libraries) and employs lot of engineer efforts to improve compatibility. However, there are still some corner cases during the interaction with uninstrumented code that CECSan did not handle well, resulting in segmentation fault. For instance, CECSan does not strip tagged pointers before passing to certain uninstrumented code. One promising direction towards better compatibility with uninstrumented programs is to combine compiler instrumentation with dynamic binary translation, where binary translation is used to instrument parts of the program that are not instrumented at compile time.

VI. Conclusion

In this paper, we propose CECSan, a highly compatible, efficient and comprehensive memory safety enhancement for C/C++ programs. Specifically, CECSan combines compiler instrumentation with a new reusable metadata table for memory safety. Additionally, CECSan also proposes to detect subobject overflows which most widely-used sanitizers fail to do, and employ various optimization to reduce performance overhead. Extensive evaluations with newly designed automation framework have clearly shown that CECSan has superior performance than representative baselines in terms of memory safety related bugs detection. Moreover, evaluation shows that our proposed sanitizer significantly outperforms the state-of-the-art in memory overhead with acceptable runtime overhead.

Acknowledgments. This project was supported by National Science Foundation of China (Grant#: 62302506).

References

[1] MITRE Corporation, "2023 CWE Top 25 Most Dangerous Software Weaknesses," Online Blog, April 2023.

[2] Dongliang Mu, "Linux Flaw Project," Github, 2019.

[3] Google Project Zero, "A Year in Review of 0-days Used In-the-Wild in 2021," Online Blog, 2022.

[4] Dokyung Song, Julian Lettner, etc, "SoK: Sanitizing for Security," IEEE Symposium on Security and Privacy (SP), pp. 1275-1295, 2019.

[5] Kostya Serebryany, AddressSanitizer: A Fast Address Sanity Checker. USENIX Annual Technical Conference, 2012.

[6] Santosh Nagarakatte, SoftBound: highly compatible and complete spatial memory safety for C. ACM-SIGPLAN Symposium on Programming Language Design and Implementation, 2009.

[7] Juan Caballero and Gustavo Grieco, Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. International Symposium on Software Testing and Analysis, 2012.

[8] George C. Necula and Jeremy Condit, CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst, pp.477-526, 2005.

[9] Evgeniy Stepanov and Kostya Serebryany, MemorySanitizer: Fast detector of uninitialized memory use in C++. IEEE/ACM International Symposium on Code Generation and Optimization, 2015.

[10] Erik van der Kouwe and Vinod Nigade, DangSan: Scalable Use-after-free Detection. Proceedings of the Twelfth European Conference on Computer Systems, 2017.

[11] Gregory J. Duck and Roland H. C. Yap, EffectiveSan: type and memory error detection using dynamically typed C/C++. ACM SIGPLAN Notices, 2017.

[12] Yuan-Fang Li, PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, 2022.

[13] Konrad Hohentanner, HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading. Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, 2023.

[14] Shengjie Xu and Wei Huang, In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021.

[15] Gregory J. Duck and Roland H. C. Yap, Stack Bounds Protection with Low Fat Pointers. Network and Distributed System Security Symposium, 2017.

[16] Myoung Jin Nam, FRAMER: a tagged-pointer capability system with memory safety applications. Proceedings of the 35th Annual Computer Security Applications Conference, 2019.

[17] Taddeus Kroes, Delta pointers: buffer overflow checks without the checks. Proceedings of the Thirteenth EuroSys Conference, 2018.

[18] Emanuel Q. Vintila, MESH: A Memory-Efficient Safe Heap for C/C++. Proceedings of the 16th International Conference on Availability, Reliability and Security, 2021.

[19] Binfa Gui, UAFSan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. Proceedings of the 30th SIGSOFT International Symposium on Software Testing and Analysis, 2021.

[20] Gregory J. Duck, Heap bounds protection with low fat pointers. Proceedings of the 25th International Conference on Compiler Construction, 2016.

[21] Yuseok Jeon, FuZZan: Efficient Sanitizer Metadata Design for Fuzzing. USENIX Annual Technical Conference, 2020.

[22] Floris Gorter and Enrico Barberis, FloatZone: Accelerating Memory Error Detection using the Floating Point Unit. USENIX Security Symposium, 2023.

[23] Yuchen Zhang and Chengbin Pang, Debloating Address Sanitizer. USENIX Security Symposium, 2022.

[24] Nathan Burow and Derrick Paul McKee, CUP: Comprehensive User-Space Protection for C/C++. Proceedings of the 2018 on Asia Conference on Computer and Communications Security, 2017.

[25] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, et al. SGXBOUNDS: Memory Safety for Shielded Execution. Proceedings of the Twelfth European Conference on Computer Systems, 2017.

[26] Thurston H. Y. Dang, Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. USENIX Security Symposium, 2017.

[27] Szekeres and Mathias Payer, SoK: Eternal War in Memory. IEEE Symposium on Security and Privacy, 2013.

[28] Santosh Nagarakatte and Jianzhou Zhao, CETS: compiler enforced temporal safety for C. International Symposium on Mathematical Morphology and Its Application to Signal and Image Processing, 2010.

[29] Kostya Serebryany and Evgenii Stepanov, Memory Tagging and how it improves C/C++ memory safety. ArXiv, 2018.

[30] Jonathan Woodruff, The CHERI capability model: Revisiting RISC in an age of risk. ACM/IEEE 41st International Symposium on Computer Architecture (ISCA),2014.

[31] Baozeng Ding, Baggy Bounds with Accurate Checking. The 23rd International Symposium on Software Reliability Engineering Workshops, 2012.

[32] Yonghae Kim, Hardware-based Always-On Heap Memory Safety. The 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020.

[33] Dongwei Chen and Daliang Xu, Saturation Memory Access: Mitigating Memory Spatial Errors without Terminating Programs. arXiv, 2020.

[34] Albert Kwon and Udit Dhawan, Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 2013.

[35] Santosh Nagarakatte and Milo M. K. Martin, Watchdog: Hardware for safe and secure manual memory management and full memory safety. The 39th Annual International Symposium on Computer Architecture (ISCA), 2012.

[36] Konrad Hohentanner, CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, 2022.