HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading

Konrad Hohentanner
konrad.hohentanner@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

Florian Kasten
florian.kasten@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

Lukas Auer
lukas.auer@aisec.fraunhofer.de
Fraunhofer AISEC
Garching, near Munich, Germany

Abstract

C/C++ are often used in high-performance areas with critical security demands, such as operating systems, browsers, and libraries. One major drawback from a security standpoint is their susceptibility to memory bugs, which are often hard to spot during development. A possible solution is the deployment of a memory safety framework such as the memory tagging framework Hardware-assisted AddressSanitizer (HWASan). The dynamic analysis tool instruments object allocations and inserts additional check logic to detect memory violations during runtime. A current limitation of memory tagging is its inability to detect intra-object memory violations i.e., over- and underflows between fields and members of `structs` and `classes`. This work addresses the issue by applying the concept of memory shading to memory tagging. We then present HWASanIO, a HWASan-based sanitizer implementing the memory shading concept to detect intra-object violations. Our evaluation shows that this increases the bug detection rate from 85.4% to 100% in the memory corruptions test cases of the Juliet Test Suite while maintaining high interoperability with existing C/C++ code.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**.

Keywords: memory safety, memory tagging, intra-object overflows, sub-object overflows, dynamic analysis

ACM Reference Format:
Konrad Hohentanner, Florian Kasten, and Lukas Auer. 2023. HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3589250.3596139

SOAP '23, June 17, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0170-2/23/06.
https://doi.org/10.1145/3589250.3596139

1 Introduction

The MITRE 2022 Top 25 Most Dangerous Software Weaknesses [5] lists memory errors in memory-unsafe languages, such as C and C++, among the leading causes for dangerous exploits. Vulnerabilities based on manual memory handling such as *out-of-bounds write/read* and *use after free* are ranked at positions 1, 5, and 7.

Sanitizers are dynamic analysis tools that can help to detect memory bugs during development by adding memory safety as part of compiler instrumentation. Hardware-assisted AddressSanitizer (HWASan) is a sanitizer available in the *gcc* and *clang* compilers, which uses memory tagging, also known as memory coloring, to improve bug detection. During object allocation, a tag is saved in the upper bits of pointers and tag memory associated with the object. On every access the tags are compared, with a mismatch leading to a runtime error.

One drawback of tagging-based sanitizers is their inability to detect intra-object violations i.e., over- and underflows between `struct` fields or `class` members, because the metadata layout handles objects as single entities without regarding the sub-objects they may contain.

In this work, we propose memory shading, a novel approach that splits the tag into color and shade. While the color part still indicates object boundaries, just like the tag in prior memory tagging approaches, the shade can now differentiate between fields and members of the same object. This allows the detection of intra-object violations and thus increases the bug-finding capabilities of tagging-based sanitizers, as we demonstrate in our prototype, HWASanIO.

In this paper, we make the following contributions:

- The memory shading concept, a modification of memory tagging, which expands the detection of temporal and spatial violations to also include intra-object violations.
- A full LLVM-based prototype implementation of HWASanIO that applies this concept to protect heap, stack, and global objects of C and C++ applications. [1]
- A detailed evaluation of HWASanIO and related solutions, specifically their effectiveness in detecting memory bugs and their performance overheads.

[1] Source code: https://github.com/Fraunhofer-AISEC/HWASanIO

2 Related Work

Compiler-based memory safety in C/C++ is possible through the use of metadata and runtime checks. This section gives an overview of available related work approaches and their respective security guarantees. Here, *spatial* memory safety ensures accesses are in-bounds, whereas *temporal* memory safety guarantees accesses are only permitted during an object's lifetime.

Interleaving *guard pages* [7, 20] or *red-zones* [4, 11, 12, 22, 24] between objects allows detection of linear overflows. Non-linear spatial violations can still occur for pointer offsets large enough to jump beyond a restricted area into another object. While these approaches can be relatively fast, they use a lot of additional runtime memory for the added restricted areas and the necessary shadowing of the application memory. They do not support intra-object detection, which would require insertion of metadata between object fields, and therefore break the Application Binary Interface (ABI).

Approaches with *bounds* metadata achieve memory safety by tracking the start and end address of objects. This can be done on a *per-object* basis, where pointer arithmetic is then instrumented to stay within the bounds of the corresponding allocations [6, 15, 21, 28], guaranteeing spatial memory safety. The precision of bounds-tracking solutions is increased with *per-pointer* metadata. *Fat-pointers* store this information directly inside the pointer [3, 14, 18, 25] to validate every spatial access. Related solutions such as Softbound/CETS keep the bounds information separated in memory and achieve temporal safety with *lock and key* metadata [16, 17, 19, 27]. Since pointers are compared to their start and end address, the check logic is quite complex. Tracing sub-object bounds is possible at the cost of an additionally increased overhead.

Some approaches deduce the object bounds from the pointer address by allocating objects in a bucketing scheme [1, 8, 10]. Because the metadata is solely defined by the object size it does not allow differentiation inside of aggregate types, and therefore offers no detection of intra-object violations. EffectiveSan [9] uses additional dynamic type information to detect spatial violations. This does include intra-object overflows, but not between fields with the same type, e.g., two `char` arrays inside a `struct`.

Memory tagging approaches [13, 23] such as HWASan use per-pointer tag metadata to achieve a low memory overhead and reduced overall check complexity compared to approaches with more complex metadata. At every access, the tag saved inside the pointer is compared for equality to the allocation tag stored in separate shadow memory directly linked to the object memory. Due to the limited tag size, metadata collisions are possible where a pointer to an object can access all other objects with the same allocation tag.

Listing 1. Intra-object overflow in Juliet CWE 121 stack-based buffer overflow testcase (variable names shortened).

```
#define STR "0123456789abcdef0123456789"
struct cV {
  char cFirst[16];
  void *vSecond;
  void *vThird; };
struct cV sCV;
memcpy(sCV.cFirst,STR,sizeof(sCV));
```

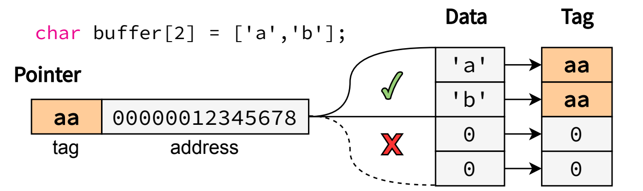
Dedicated hardware can be used to lower the runtime overheads of memory safety approaches. CHERI [26] uses so-called capabilities, i.e., fat pointers in hardware, to store bounds information with optional support for sub-bounds tracking, but requires considerable integration effort. ARM Memory Tagging Extension (MTE) [2] is a hardware-assisted memory tagging approach introduced as part of ARMv8.5. Tags are saved directly in hardware, with new instructions used for tag access. The load and store logic is able to automatically compare these tags, which greatly improves the performance compared to HWASan. The coarse granularity of tagging every 16 bytes of memory with only 4 bits of metadata leads to a higher collision rate between objects. Since the hardware extension only supports a single tag per object, no intra-object detection is possible.

In summary, current memory safety solutions only sparingly detect intra-object violations, with no compiler-based memory tagging solution being able to detect them.

3 Intra-object Memory Violations

An intra-object, or sub-object, memory violation occurs due to an over- or underflow between fields of the same object. An analysis of the source code used in the SPEC CPU 2017 benchmark, which includes compilers and image processing applications, shows that only 10% of all used aggregate variables are buffers, and 90% are `structs` and `classes`. With an average of 9 fields contained in these variables, the potential for intra-object violations is high. For current memory-tagging-based sanitizers such as HWASan, a detection is not possible because objects are seen as one unit during allocation with no differentiation between sub-objects.

Listing 1 shows a code snippet of a memory corruption test case from the Juliet test suite. Because the `sizeof` calculation is based on the whole `struct` and not the 16 byte buffer, the `memcpy` function is called with a too large size for its target. This results in an intra-object overflow and the corruption of all other data fields. By carefully overwriting neighboring variables, it is possible to either manipulate application data directly or, if the variables contain function or data pointers, start hijacking attacks.



```
char buffer[2] = ['a','b'];
```

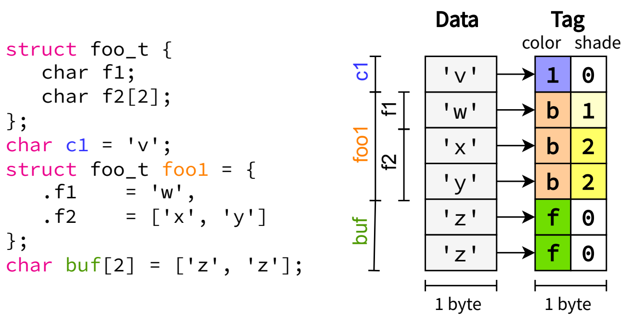
Figure 1. In memory tagging, tags are saved in pointer and object metadata to identify legal and illegal accesses.

4 Memory Tagging and HWASan

Memory tagging, or memory coloring, is a mechanism that can be used to enforce memory safety guarantees in an application by adding instructions to track a per-object tag in both the pointer address and dedicated metadata memory. Figure 1 shows this fundamental concept. During allocation, the buffer variable is assigned a random tag that is different from the previous and next memory object. The tag is added across the shadow memory range corresponding to the memory address of the buffer variable, as well as in the upper bits of the pointer. Every time this pointer is dereferenced, the two tags are compared to ensure accesses are in-bounds. The memory tagging implementation of HWASan included in the LLVM compiler infrastructure uses a flexible *granularity*, which refers to the number of bytes tagged with the same tag. In the default 16-to-1 mapping of HWASan, 16 bytes of application memory are associated with one byte of metadata. To be able to set a byte-precise end address inside this 16 byte range, a granule byte is added at the end of an object's metadata to indicate the last legally accessible byte. This requires special handling in the check logic. The metadata is saved in dedicated shadow memory mapped in the virtual address space; its location is derived from the application memory address. Pointers themselves include the tag in the 8 upper pointer bits, which are unused in virtual addressing and do not interfere with load and store instructions due to the ARM Top Byte Ignore (TBI) hardware feature.

From the application's point of view, the buffer variable is used normally during all operations, with the special case of load and store operations. Here, the compiler instrumentation adds instructions before each memory access to load the tag from the shadow memory address linked to the target address and compare it for equality with the tag stored inside the pointer. If the tags match, the operation continues normally. In the case of a mismatch, e.g., dereferencing a pointer incremented beyond the object's end address, an exception is thrown to indicate and prevent the memory violation. Additional debugging information, such as the stack frame and violation type, help to find the root cause of the memory corruption.

Memory tagging also allows the detection of temporal violations, because the shadow memory metadata of freed objects is set to a zero tag, invalidating all pointers to the object. Pointers to the now unallocated objects can therefore not be dereferenced anymore, preventing violations such as *use after free*.



```
struct foo_t {
   char f1;
   char f2[2];
};
char c1 = 'v';
struct foo_t foo1 = {
   .f1    = 'w',
   .f2    = ['x', 'y']
};
char buf[2] = ['z', 'z'];
```

Figure 2. Memory shading splits the tag into color and shade to differentiate between objects and object fields.

5 Memory Shading

Memory shading expands the idea of memory tagging to offer protection against intra-object memory violations. To achieve this, the metadata design is changed from a purely color-based approach to a color- and shading-based approach, where the shade is used to differentiate between object members. Note: The shading concept covers both `structs` and `classes`, and both are protected in our prototype. To avoid redundancies we use `structs` in the following explanations and examples.

Color. The *color* is defined by the upper half of metadata. It is used in the same manner as the tag described in Section 4. During an object's allocation, it is set to a random non-zero value that does not collide with previous and following objects, which is ensured by comparing it with the color of neighboring metadata and generating new values until a differing value is found. The color is the same across the entire shadow memory range corresponding to the object and is used in all of the object's load and store checks.

Shade. The remaining bits of metadata are used as the *shade* of an object. It is different for neighboring fields, and thus allows detection of intra-object under- and overflows. For non-`struct` objects, it is set to zero, because any pointer to the object can legally access all its allocated memory. For `struct` objects, the shade starts at value one and is incremented for every additional non-`struct` field, wrapping around back to one after it reaches its maximum value. For layered `structs`, where a field is of a `struct` type, the shading algorithm is initiated anew, starting the shade of the field at value one. This keeps the shading layout consistent for objects with an arbitrary depth.

Example. Figure 2 shows an exemplary runtime memory layout of two variables on a system with the memory shading concept active. The C `struct` type `foo_t` contains a char and a char buffer field. The metadata consists of a shared tag across the object and individual shades for each of the field variables. Additionally, a char and a buffer variable are allocated, which are not of a `struct` type and therefore have their shade set to zero, i.e., with no differentiation inside the object. Here, over- and underflows to other objects are prevented by the color stored in the metadata.

6 HWASanIO Implementation

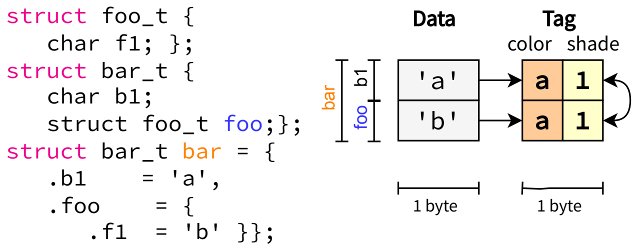
The HWASanIO prototype is implemented as an extension to the HWASan source code available in the LLVM compiler framework (version 14), with support for Linux-based systems running on ARMv8 hardware. The following modifications are needed to add the support for memory shading to the HWASan instrumentation pass and its runtime library.

Metadata generation. Metadata in HWASanIO is generated with 4 bits for both color and shade, as discussed in Section 5. Here, the interceptors for heap management functions, as well as the stack and globals metadata handling have been adapted accordingly. Additionally, the metadata generation was modified to prevent tag collisions for neighboring objects by comparing against the previous and following tags in shadow memory. This ensures the detection of linear overflows and underflows.

Shadow Memory Layout. HWASan uses a 16-to-1 mapping, where every block of 16 bytes of data is shadowed by one byte of metadata. Object sizes therefore have to be increased to multiples of 16 bytes by adding padding bytes. This approach is not desirable for memory shading, because `struct`s with additional padding between the fields would break the ABI and therefore dynamically linked uninstrumented library interfaces. Thus, HWASanIO uses a 1-to-1 mapping without padding. Shadow addresses are derived by flipping the Most Significant Bit (MSB) of the user-space virtual address.

Pointer instrumentation. An important aspect of color shading is the continuous update of the pointer shade. In HWASanIO, deriving a pointer to a sub-object sets the shade according to the targeted field. Here, the instrumentation pass inserts instructions for setting the corresponding shade whenever a sub-object pointer is created.

Check logic and optimizations. The check logic was modified to compare both color and shade stored in pointers and shadow memory, which leads to additional instructions being added. To reduce this overhead, new compile-time optimizations are introduced. A pointer flow analysis traces allocated objects that are never cast to `struct`-types and can therefore use the shorter, color-only, check on memory accesses. If an object's origin cannot be safely determined, the shade-aware check logic is used to avoid false negatives. This is the case for e.g., memory handling functions that take `void*` arguments and are used with both `struct` and non-`struct` arguments. For the SPEC CPU 2017 benchmark, this analysis safely simplified an average of 51% of checks to a color-only check.



```
struct foo_t {
   char f1; };
struct bar_t {
   char b1;
   struct foo_t foo;};
struct bar_t bar = {
   .b1    = 'a',
   .foo    = {
      .f1  = 'b' }};
```

Figure 3. Shade collision in a layered `struct`.

Granule. HWASan uses a granule byte to mark the end address of an object on a byte-level granularity as discussed in Section 4. Because HWASanIO uses a 1-to-1 mapping and therefore always guarantees byte precision, the feature was removed without impacts to the memory safety guarantees.

Reporting. The reporting functionality of HWASan was enhanced to handle intra-object errors. The output contains the violation type and memory location of the corresponding variables as well as the usual debugging information such as stack trace and source code location.

Standard library wrappers. Compared to HWASan, the HWASanIO source code contains additional standard library wrappers for common memory handling functions such as *strcpy* to check the validity of the start and end of the source and destination arguments.

7 Security Discussion

In this section, we discuss the cases where HWASanIO is unable to detect memory violations during runtime, i.e., false negatives.

Due to the tag length, in some cases *tag collisions* are possible. While HWASanIO guarantees that neighboring allocations are assigned different colors, two non-neighboring objects may share a tag with a probability of approximately 7%. In this case, a non-linear access through a pointer to another object with the same tag is possible. This also applies to shade collisions of non-neighboring fields.

Collisions can also occur as a result of the algorithm assigning the shading values. To keep a consistent shade layout between all objects of the same `struct` type, field shades always start with value one and are then incremented. In the case of nested `struct`s, this may lead to a shade collision for some layouts thereby preventing the detection of intra-object violations at either or both ends of a sub-object. An example is depicted in Figure 3. The `struct` variable `bar` contains a `char` variable and a `struct` field which consists of a single `char` variable. In this case, a neighboring over- or underflow between the fields, indicated by the arrow, cannot be detected, because the fields share the same shade.

Table 1. Juliet Test Suite vulnerability detection ratio.

CWE (# cases)	HWAIO (5364)	ASan (5364)	HWASan (5364)	Eff.San (5364)	Softb. (3970)
Stack Overflow	100%	96.7%	82.9%	55.8%	77.7%
Heap Overflow	100%	94.7%	94.6%	58.0%	73.7%
Buf. Underwr.	100%	100%	81.9%	100%	82.5%
Buf. Overrd.	100%	100%	99.7%	62.1%	96.5%
Buf. Underrd.	100%	100%	75.9%	98.0%	78.4%
Double Free	100%	100%	100%	0%	100%
Use After Free	100%	83.0%	50.9%	5.4%	51.3%
Free Inside Buf.	100%	100%	0%	0%	100%
Overall	100%	97.0%	85.4%	62.3%	78.9%

8 Evaluation

We evaluated both the bug detection capabilities and the performance and memory overheads of our HWASanIO prototype and approaches from related work. We compared our results with measurements of the unmodified HWASan [23] and the sanitizers ASan [22], EffectiveSan [9], and Softbound/CETS [16, 17]. They represent viable solutions for sanitizers using memory tagging, red-zones, dynamic typing, and bounds-checking, respectively. The tests were executed on an M2 MacBook Pro 2022 running Debian (Linux 5.19), except for EffectiveSan and Softbound/CETS which do not support ARM and were therefore evaluated on an AMD EPYC 7742 x86_64 system running Debian (Linux 5.10). Performance and memory overheads were measured with the SPEC CPU 2017[2] benchmark suite. For the functional evaluation, we used the Juliet Test Suite[3].

8.1 Vulnerability Detection Rate

The Juliet Test Suite is a collection of C/C++ test cases using Common Weakness Enumerations (CWEs) based on real-world applications. The test suite contains many temporal and spatial memory corruptions, which are always provided as a *good* version without any memory bugs and a *bad* version containing the memory corruption. Multiple variants of each test case are provided, where the same vulnerability exists in different control flow or data flow constructs, e.g., embedding the memory bug behind an *if*-condition.

For our evaluation, we removed variants which are difficult to automate because they depend on external or random input and only occasionally or never create an actual memory bug at run-time, resulting in a total of 5364 test cases. For the evaluation of Softbound/CETS, we excluded test cases that lead to compilation errors, resulting in a total of 3970 cases.

Table 1 lists the results of evaluating HWASanIO and related solutions with the Juliet Test Suite. None of the approaches showed crashes in the good variants of the test cases.

Importantly, HWASanIO is the only sanitizer with complete detection of all temporal and spatial bugs, and achieves a 100% vulnerability detection rate. AddressSanitizer (ASan) also achieves a very high detection rate of 97% and only misses violations in the intra-object overflow test cases and some *use after free* cases. Here, the Juliet Test Suite does not contain additional categories which would further differentiate the detection rate of HWASanIO from ASan such as non-linear overflows or additional variants of intra-object corruptions. Besides HWASanIO, only EffectiveSan can detect intra-object overflows between fields of differing types. Although Softbound/CETS has the conceptual support for it, the available implementation is incomplete in this regard. For HWASan, missing wrappers and an incomplete temporal bug detection account for most of the non-mitigated bugs. The dynamic typing used in EffectiveSan cannot detect many vulnerabilities in the overflow categories, because the test cases use buffers of the same type. While the bounds tracking used in Softbound/CETS should be able to detect most bugs, the released source code lacks features and is missing various wrapper functions.

In summary, HWASanIO is the only sanitizer able to detect all memory violations in the Juliet Test Suite. The Juliet Test Suite results therefore confirm the high bug detection capabilities of memory shading in practice.

8.2 Performance and Memory Overhead

To measure HWASanIO's overhead, we used the SPEC CPU 2017 benchmark suite, comparing the performance and memory consumption of benchmarks compiled with HWASanIO and other solutions to an uninstrumented baseline of the benchmarks. We compiled all benchmarks with optimization level `O2` and executed them in single-threaded mode. Out of the 17 C/C++ *rate* benchmarks, we had to exclude *502.gcc* and *523.xalancbmk*, because of a memory violation detected by all sanitizers and *526.blender* due to undefined behavior detected by HWASanIO and EffectiveSan.

Softbound/CETS was ported from LLVM 3.9 to LLVM 9 to compile and run SPEC CPU 2017. Not all tests could be evaluated due to errors in the instrumentation.

Performance. The results of our performance evaluation are shown in Table 2. The evaluation shows that HWASanIO achieves an average performance overhead of 242.8%, which is around 90% slower than HWASan. When looking at the geometric mean, HWASanIO achieves a 129.1% overhead, which is close to the 118.9% overhead of HWASan. The additional overhead is largely due to the runtime of the `511.povray` benchmark, where the allocation of large numbers of local `struct` variables slows down the execution with HWASanIO. In the benchmark, the *Ray_In_Bound* function is executed an average of $8.12x10^{10}$ times per second and allocates a `struct` with 26 fields every time. For each of these fields a call to the shading function of HWASanIO is

[2] https://www.spec.org/
[3] https://samate.nist.gov/SARD/test-suites/112

Table 2. Measured performance overhead of memory safety frameworks for SPEC CPU 2017 benchmarks.

	Performance Overhead					**Memory Overhead**				
Benchmark	**HWASanIO**	**HWASan**	**ASan**	**Eff.San**	**Softb.**	**HWASanIO**	**HWASan**	**ASan**	**Eff.San**	**Softb.**
500.perlbench	367.8%	302.2%	90.4%	713.7%	-	146.8%	26.3%	429.1%	89.1%	-
505.mcf	81.8%	83.1%	38.6%	236.8%	584.9%	147.9%	6.6%	47.8%	3.5%	638.0%
508.namd	191.8%	260.9%	73.6%	131.5%	350.8%	110.5%	9.4%	204.9%	117.1%	88.3%
510.parest	185.3%	215.9%	66.8%	551.7%	-	154.5%	30.1%	458.5%	84.6%	-
511.povray	1607.6%	345.5%	153.6%	1284.7%	-	261.2%	115.0%	1018.8%	1499.8%	-
519.lbm	82.4%	37.0%	9.2%	131.0%	148.0%	100.1%	7.7%	13.8%	2.3%	1.9%
520.omnetpp	154.9%	88.4%	99.4%	203.6%	-	119.3%	27.4%	391.7%	71.6%	-
523.xalancbmk	127.3%	104.3%	69.4%	354.7%	-	148.2%	31.2%	563.7%	44.0%	-
531.deepsjeng	112.7%	127.2%	45.5%	90.9%	510.7%	99.9%	6.5%	1.3%	2.1%	2.2%
538.imagick	223.8%	204.9%	90.1%	195.8%	609.7%	133.2%	7.9%	186.2%	13.6%	106.5%
541.leela	133.0%	103.6%	62.1%	202.8%	-	100.2%	41.9%	7784.1%	387.1%	-
544.nab	84.5%	128.4%	43.9%	58.1%	243.4%	139.5%	29.0%	274.1%	70.1%	40.7%
557.xz	32.6%	49.4%	19.4%	78.0%	-	96.4%	4.4%	32.3%	3.8%	-
999.specrand	13.8%	39.5%	95.5%	12.4%	46.1%	154.0%	150.4%	444.5%	164.9%	390.9%
Average	242.8%	149.3%	68.4%	303.3%	356.2%	136.6%	35.3%	846.5%	182.4%	181.2%
Geometric Mean	129.1%	118.9%	56.6%	177.3%	270.5%	131.7%	19.9%	182.0%	38.6%	45.4%

required. A possible optimization to combat this issue is the preparation of the shading layout for the entire `struct` during instrumentation at the cost of an increased binary size instead of initializing every field individually. When looking at other benchmarks such as `557.xz`, HWASanIO runs faster than the original HWASan. This can be explained by the faster metadata access of the 1-to-1 metadata mapping as well as the simpler non-granule check logic for non-`struct` objects (see Section 6).

Memory Overhead. To compare the memory overhead we measured the peak memory consumption of the instrumented benchmarks with the GNU `time` command and compared the measurements with the non-instrumented baselines. Table 2 shows the average memory overheads of the evaluated approaches.

HWASanIO achieves an average memory overhead of 136.6%, compared to a 35.3% overhead for the original HWASan. This corresponds to the difference in mapping granularity used, where HWASanIO uses 1 byte of metadata per application byte and HWASan shadows 16 application bytes with a single tag byte. Since the 1-to-1 mapping is needed to conform with the ABI and the 8-bit metadata is already quite small, a reduction of the memory overhead is not possible.

Comparison. HWASanIO expands the bug detection capabilities at the cost of an additional memory and performance overhead. From a runtime performance perspective, the memory shading metadata management does not considerably impact the overall runtime behavior in most cases, except for applications that use large amounts of local `struct` variables. In comparison, ASan achieves lower runtime overheads than HWASanIO, which can be explained by its very efficient metadata setup and check logic, but it does not detect intra-object violations and has a larger memory overhead. The base version of HWASan has a significantly lower memory overhead but does not detect as many bugs, making it a good sanitizer choice for resource-constrained devices. EffectiveSan, the only other sanitizer which can detect intra-object violations between fields of different types, is slower than HWASanIO and requires more memory. Softbound/ CETS is outperformed by HWASanIO in both memory and performance overhead. In summary, the new analysis capability is achievable with a manageable overhead increase.

9 Conclusion

We proposed the memory shading concept for C/C++ programs to allow intra-object detection for memory-tagging-based sanitizers. We implemented the HWASanIO prototype as an extension to HWASan, providing memory safety by protecting a program's heap, stack, and globals against both spatial and temporal memory corruptions and offering efficient intra-object detection. Since HWASanIO's design does not affect the underlying ABI it achieves a high interoperability level supporting existing C/C++ source code. We implemented a fully functional, LLVM-based HWASanIO prototype for ARM-based Linux systems which is available as compiler flag. In our security evaluation, the HWASanIO prototype detects more bugs than similar software-based solutions while maintaining a manageable performance and memory overhead for dynamic analysis. In summary, the HWASanIO prototype is a viable dynamic analysis tool to efficiently detect memory corruptions in C/C++ programs, including the previously hard-to-detect intra-object violations.

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium (SEC '09)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/baggy-bounds-checking-efficient-and

[2] ARM Limited. 2019. *ARM Architecture Reference Manual – ARMv8-A, for ARMv8-A architecture profile.* ARM DDI 0487E.a (ID070919).

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/178243.178446

[4] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE, Washington, DC, USA, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[5] MITRE Corporation. 2022. 2022 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. Accessed: 2023-03-15.

[6] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 162–171. https://doi.org/10.1145/1134285.1134309

[7] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN '06)*. IEEE. https://doi.org/10.1109/DSN.2006.31

[8] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[9] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. *SIGPLAN Not.* 53, 4 (jun 2018), 181–195. https://doi.org/10.1145/3296979.3192388

[10] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17)*. Internet Society. https://doi.org/10.14722/ndss.2017.23287

[11] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-Weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[12] Reed Hastings and Bob Joyce. 1991. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 USENIX Conference*.

[13] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. 2023. CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3555776.3577635

[14] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*. USENIX Association, 275–288.

[15] Richard W M Jones and Paul H J Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the 3rd International Workshop on Automatic and Algorithmic Debugging (AADEBUG '97)*. Linköping University Electronic Press.

[16] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[17] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[18] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 128–139. https://doi.org/10.1145/503272.503286

[19] Harish Patil and Charles Fischer. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practics and Experience* 27, 1 (Jan. 1997).

[20] Bruce Perens. 1999. Electric Fence Malloc Debugger. https://linux.die.net/man/3/libefence. Accessed: 2023-03-15.

[21] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*. Internet Society.

[22] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[23] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and How it Improves C/C++ Memory Safety. arXiv:cs.CR/1802.09517 https://arxiv.org/abs/1802.09517

[24] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC) (ATC '05)*. USENIX Association.

[25] Joseph L. Steffen. 1992. Adding Run-Time Checking to the Portable C Compiler. *Software: Practice and Experience* 22, 4 (1992).

[26] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2015.9

[27] Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/1029894.1029913

[28] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/1755688.1755707

Received 2023-03-10; accepted 2023-04-21