METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening

Istvan Haller
Vrije Universiteit Amsterdam
i.haller@student.vu.nl

Erik van der Kouwe
Vrije Universiteit Amsterdam
vdkouwe@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

ABSTRACT

Many systems software security hardening solutions rely on the ability to look up metadata for individual memory objects during the execution, but state-of-the-art metadata management schemes incur significant lookup-time or allocation-time overheads and are unable to handle different memory objects (i.e., stack, heap, and global) in a comprehensive and uniform manner.

We present METAlloc, a new memory metadata management scheme which addresses all the key limitations of existing solutions. Our design relies on a compact memory shadowing scheme empowered by an alignment-based object allocation strategy. METAlloc's allocation strategy ensures that all the memory objects within a page share the same alignment class and each object is always allocated to use the largest alignment class possible. This strategy provides a fast memory-to-metadata mapping, while minimizing metadata size and reducing memory fragmentation. We implemented and evaluated METAlloc on Linux and show that METAlloc (de)allocations incur just 3.6% run-time performance overhead, paving the way for practical software security hardening in real-world deployment scenarios.

1. INTRODUCTION

Many common software security hardening solutions need to maintain and look up memory metadata at runtime. Examples include bounds information to validate array references [1,2], type information to validate cast operations [12], solutions that prevent use-after-free exploits [11,15], and object pointer information to perform garbage collection [13]. While newer programming languages can often track such metadata in-band using fat pointers, previous efforts to implement in-band metadata management in systems programming languages, such as C or C++, have found limited applicability due to poor ABI compatibility and nontrivial overhead [4].

The alternative solution is to associate metadata information with the memory objects themselves, assuming we have a mechanism to map pointers to the appropriate metadata. Such a primitive is the key to implementing modern metadata management schemes, but it is also challenging because it needs to support all the possible memory objects (heap objects allocated with `malloc()`, globals, and stack objects) as well as minimize the performance and memory impact of metadata update and lookup operations.

Minimizing the impact of update operations is challenging, because allocation and deallocation of memory objects and their metadata occurs frequently during execution. Minimizing the impact of lookups is also challenging, since metadata lookup must be able to support interior pointers into nested classes, structures, and arrays—dictating support for range queries and disqualifying the use of space- and time-efficient hash tables.

Current state-of-the-art software hardening projects all rely on tailored, mostly one-off solutions for metadata management, but none of them simultaneously achieves low lookup and update impact in all cases. As a result, none of them provides a generic solution. Common approaches include tree-based metadata handling and memory shadowing.

Tree-based approaches [7, 11] store an interval node for each allocated object according to its bounds. Unfortunately, tree lookups can result in a prohibitive performance hit, as the tree depth is frequently in the double digit range (more than 1,024 memory objects). The lookup time is also unpredictable, as it varies with the object count. As a result, tree-based systems are unsuitable for most production situations.

EUROSEC'16, April 18-21 2016, London, United Kingdom

 ISBN 978-1-4503-4295-7/16/04…$15.00

DOI: http://dx.doi.org/10.1145/2905760.2905766

Traditional memory shadowing, in turn, relies on a fixed pointer-to-metadata mapping [1, 2, 15]. The key design choice for this approach is the *metadata compression ratio*. The metadata compression ratio represents the number of metadata bytes that need to be tracked for each data byte. For example, assume we store one byte of metadata for each block of eight bytes. In this case the compression ratio is $\frac{1}{8}$. If we have a pointer p and an array of metadata starting at address q, we can compute a pointer to the metadata for object as $\frac{1}{8}p + q$. This way, metadata can be located very efficiently. However, choosing the appropriate compression ratio is difficult, as it enforces a minimum alignment on every memory allocation. Small compression ratios result in inflated metadata size and a large tracking overhead, while large compression ratios result in significant memory fragmentation. In practice, memory management systems typically only guarantee alignment up to 8 or 16 bytes. This means that to keep the compression ratio reasonably small only a single byte of metadata is supported [1,2]. Even then, this approach introduces prohibitive initialization time and memory overhead for large objects in case multi-byte metadata is needed. Finally, recent approaches rely on custom allocators to reduce the impact of memory shadowing on the heap, but cannot support efficient and comprehensive metadata management including more performance-sensitive objects on the stack [12].

In this paper, we propose METAlloc, a new metadata memory management scheme based on an efficient and comprehensive variable memory shadowing strategy. Our strategy builds on recent developments in heap [6] and stack [10] organizations to implement a variable and uniform pointer-to-shadow mapping and significantly reduce the performance and memory impact of metadata management. Our results show that METAlloc is practical and can support efficient whole-memory metadata management for several software security hardening solutions.

Summarizing, we make the following contributions:

- We propose a new memory metadata management scheme that supports interior pointers and is time- and space-efficient in both lookups and updates across all memory object types.
- We present a prototype implementation termed METAlloc, which demonstrates that efficient and comprehensive metadata management is feasible and widely applicable in practice.
- We present an empirical evaluation showing that METAlloc incurs a run-time performance overhead of just 3.6% for (de)allocations on SPEC2006.

2. METAlloc

As we have seen, one of the major limitations of state-of-the-art memory shadowing approaches is the difficulty of getting the compression ratio right. Because the right value may differ from application to application, the intuitive solution is to enable a variable compression ratio. This eliminates the fixed memory overhead associated with metadata shadowing and greatly reduces the allocation-time performance hit.

METAlloc's key goal is to implement a metadata management scheme handling all memory objects in a uniform and highly efficient manner, regardless of their allocation type (heap, stack, or global memory). There are two requirements to accomplish this goal. The first is to support a simple, efficient, and uniform mechanism to associate pointers with the compression ratio. The second is the ability to optimize the compression ratio as much as possible, ideally such that only a single metadata entry is needed for each object. METAlloc meets both these requirements by ensuring that all the memory objects within a memory page share a nontrivial common alignment, which is fixed as long as there are active objects within the page. This requirement serves as a basis for our scheme and is met by drawing from modern heap [6] and stack [10] organizations widely used in production, as discussed in Section 2.4.

Alignment relates directly to the compression ratio, namely an n-byte object alignment allows one *metadata entry* to be associated to every group of n bytes within said object. Having uniform alignment within each memory page allows METAlloc to associate compression ratios to the individual memory pages and to look them up using a mechanism similar to *page tables*. Such page tables also include the location of the metadata region corresponding to the individual pages. This mechanism is described in the following section.

2.1 Efficient retrieval of page information

Because lookups are expected to be very frequent, the page table design is very performance-sensitive. For this reason, METAlloc opts for a single-level page table design, which requires only one memory read for each lookup. We refer to this data structure as the *meta-page table*. Figure 1 shows the data structures of METAlloc, including the use of the meta-page table. Given a pointer, we split it in a page index and an offset. The page index is used as an index into the meta-page table, which is an array stored at the page table base. This page table base is a compile-time constant and therefore requires no extra memory read. The entries in this array are eight bytes large, split between the seven-byte *metabase* pointer and the one-byte base-2 logarithm of the memory alignment. The metabase points to the start of an array of metadata entries available for this page table entry. The size of the entries of the metadata array is determined by the *metasize* compile-time constant, which specifies the amount of metadata per object. It should be noted that objects larger than the alignment size have multiple metadata entries, each with the same contents. The offset part of the original pointer divided by the alignment serves as an index into the metadata array, allowing the correct metadata

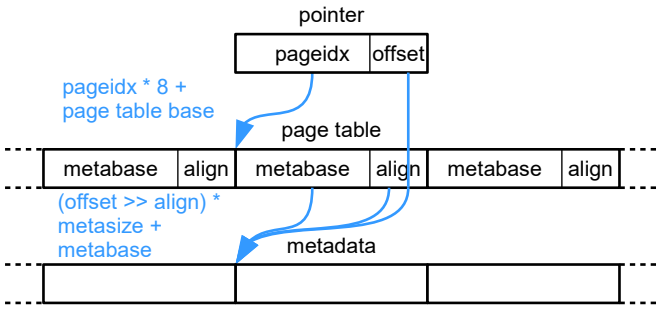

Figure 1: METAlloc's data structures

entry to be located for the specified object.

While the size of the meta-page table (for x86-64 systems using 48-bit virtual addresses and 4096-byte pages) is theoretically 2^{36} entries or 512 GB, only pages corresponding to address ranges currently in use by the process need to be allocated. To implement this strategy, METAlloc reserves the required virtual memory area in advance and relies on demand paging to lazily link the required page table pages to physical memory.

2.2 Static versus dynamic metadata

The scheme presented so far assumes a fixed-sized metadata entry which is statically initialized at the start of the object's lifetime. For objects whose size exceeds their alignment there are multiple copies of this metadata entry in memory. A metadata entry can be an integer, a small struct or even a pointer, which can refer to further metadata of arbitrary dynamic size.

In the simplest possible setup, each metadata entry is filled with a compile-time constant, such as a type index. This scheme allows the instrumentation to identify certain predetermined object characteristics at run time, with the lowest possible overhead. In this use case, initializing the metadata is cheap, as it involves a simple `memset` operation for the compressed metadata range corresponding to the object. Given the use of variable compression rates, the amount of metadata that needs to be initialized is minimal.

Alternatively, the metadata can include a pointer to information created at compile time in global memory. This scheme may be, for example, used to support in-depth type tracking when implementing precise garbage collection for languages such as C/C++ [13]. From an overhead perspective, this is similar to the constant metadata scenario, as the pointer itself is just another numeric constant, whose value is fixed at compile time.

Finally, the metadata pointer might be generated at run time. In this case, a *metadata node* is allocated whenever a new memory object is allocated and deallocated when it is freed. This scheme is relevant to instrumentation tailored to individual object instances, rather than broad object groups. Sharing lifetimes is trivial on the stack and in global memory, but leads to interesting design decisions when dealing with the heap. Intuitively, the fastest solution is to increase the allocated size of the object itself and store the metadata node in-band. In practice, we found this solution to result in significant performance degradation due to its negative caching impact. The alternative is to perform a second heap allocation dedicated to the metadata node itself. While we expected a nontrivial performance hit for applications where allocations dominate the run time, we found the impact to be reasonable in practice. In addition, most applications do not require object-specific metadata tracking and can safely refrain from using dynamic metadata, as it incurs the highest run-time and memory overhead.

2.3 Instrumentation across memory types

Globals. Global memory is the simplest to deal with from a memory shadowing perspective. Global allocations only occur at load time and the amount of global data is typically limited. These properties suggest that the default 8-byte alignment within the system works reasonably well for this allocation type, thus we can just associate every global data memory page with this alignment in the meta-page table. We simply instrument binary files (executables and dynamic libraries) to allocate metadata pages and set up the meta-page table entries corresponding to their data sections as soon as the binaries are loaded.

Stack. Stack pages share the property of long lifetimes with global memory (associated with the stack for the lifetime of the thread), but they are unique in that object allocations within the pages occur frequently and with minimal run-time cost. A simple solution is to restrict stack objects to the conservative 8-byte alignment [1], but this makes tracking multi-byte metadata prohibitively expensive. METAlloc solves this challenge by leveraging recent advances in shadow stack solutions (recently integrated in mainstream compilers) [10], which split the program stack into a primary and a secondary stack. METAlloc uses a multi-stack approach with a primary and a number of secondary stacks. The primary stack preserves all the stack objects not subject to the current instrumentation, including ABI specific elements such as return addresses and arguments, while the secondary stacks store the remainder. The secondary stacks each are designed for a particular class of object sizes with appropriate non-trivial alignment to improve the compression ratio for metadata tracking. This design ensures that METAlloc only needs to care about the secondary stacks, which are free from ABI specific restrictions. We propose using the same heuristic for all instrumentations, namely moving all objects which can potentially be subject to memory corruption attacks (either address taken or address used in unpredictable manner) to one of the secondary stacks. In practice the secondary stacks end up composed mostly of arrays and some address taken integers. As a result, we propose enforcing a relatively large alignment on each object within every secondary stacks. While

some memory fragmentation does occur with address taken integers, it only affects a small portion of the entire memory address space of the program and it will not break program functionality in general. Our current design uses one secondary stack for small and medium objects having a fixed alignment of 64 bytes, and another secondary stack for large objects where 4096 byte alignment is enforced. METAlloc instruments the allocation of program stacks to create the secondary stacks and to also allocate the corresponding metadata pages and to set up the appropriate meta-page table entries.

Heap. In order to meet our requirements, METAlloc uses a heap allocator designed around the concept of object sizes. Instead of keeping track of the heap as a whole, it operates with size-specific free-lists instead. Whenever a free-list becomes empty, the allocator can request new memory pages from the system, which are then associated with this particular free-list until they are released back to the system. The allocator would then enforce the largest possible alignment for objects within each free-list without triggering too much fragmentation. Applying METAlloc to such a heap allocator is trivial as one just needs to monitor page request by the free-lists to associate the appropriate metadata information with the pages. Figure 2 summarizes this operation with the addition of potential object caches and a page allocator to improve performance.

2.4 Implementation specifics

While it is possible to build a new custom heap allocator which respects our design specification, we decided to build upon a proven, state-of-the-art allocator instead. We expect that for most complex C++ applications the heap allocator has a significant impact on performance, thus a proven allocator is key. The tcmalloc allocator (developed and used by Google) features a memory organization which matches our requirements. Neither tcmalloc nor our modifications affect ABI compatibility, so no changes are needed in the operating system and external libraries do not need to be recompiled.

For the stack we decided to extend the implementation of SafeStack [10], as it is advertised as a replacement for stack canaries going forward and it is becoming a core component within the LLVM project. Its interpretation of safe and unsafe objects also matches up well with our definition of objects requiring instrumentation. SafeStack does not affect ABI compatibility.

3. APPLICATIONS

Efficient metadata tracking enables a wide range of valuable instrumentation tools to be used with production systems where performance overhead is a key characteristic. In the following we present a couple of key examples of such instrumentation. This list is by no means exhaustive and we hope that readers find other innovative uses of the framework. In this (short) paper, the applications serve as motivation for our work and

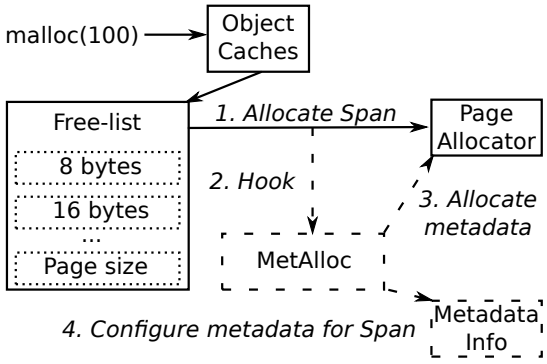

Figure 2: Heap metadata management using METAlloc

we will focus our evaluation on the framework itself.

3.1 Write Integrity Protection

Recent developments in attack techniques [3, 5, 14] show the need to enforce additional data integrity within the program besides the classic control-flow integrity. Both Microsoft's WIT [1] and Oracle Application Data Integrity have looked into the topic of restricting the target addresses of memory writes using a coloring mechanism. In these schemes each memory location is associated with a given color and the instrumentation alongside each memory write checks if the target location matches the color of the pointer/instruction. In the case of both of these systems, the color for the memory location is tracked using metadata shadowing with a fixed compression ratio. Replacing these systems with METAlloc can lead to substantial improvements in allocation performance.

3.2 Bounds Checking

Efficient bounds checking has been proposed in the past to counter buffer overflow vulnerabilities, but none of the solutions ended up in production systems due to the performance and memory overheads they bring. One particularly efficient example is Baggy Bounds Checking [2] which offers a strong protection model with limited memory overhead, based on fixed compression metadata. Its primary deficiency is the need to allocate objects in slots with sizes in the powers of two, a requirement that is typically not enforced in generic heap allocators due to the potential for high internal fragmentation. The system can be rebuilt without the alignment requirements but that would require tracking base pointer and size information for every object, which leads to performance and memory issues with the fixed compression ratio (it is prohibitive to store 16 bytes of metadata for every 8 data bytes). METAlloc and its variable compression ratio can help to deal with the problematic large object allocations, ensuring consistently low overhead across applications even when using multiple metadata bytes.

An alternative implementation of bounds checking is Light-weight Bounds Checking [8]. This system detects

out-of-bounds accesses at the memory access time instead of during the pointer arithmetic. The system injects guard zones between objects and fills them with a random byte value to detect any access into these regions. A memory access is safe if it returns a different value, but real data might also accidentally match the guard value. An additional check is performed in the latter case to filter out false positives, but on average it is only performed with probability 1 in 256. This check retrieves a metadata bit associated with the address which specifies if it belongs to real data or one of the guard zones. Light-weight Bounds Checking uses a fixed compression ratio shadowing scheme of one metadata bit for every byte of data in the program. However, metadata retrieval is avoided on the fast-path of this scheme with little impact on performance. As a result replacing the existing metadata tracking with METAlloc only yields benefits to the system. The existing system uses a hierarchical metadata storage system requiring two memory accesses to retrieve the metadata bit. METAlloc also performs two memory accesses, but it involves more pointer arithmetic instructions. It is safe to say that the fast-path behaviour will easily hide the small difference in retrieval overhead. On the other hand the variable compression ratio of METAlloc reduces allocation overhead, which can be significant in many applications. As a result, Light-weight Bounds Checking can also benefit from using METAlloc for its metadata tracking.

3.3 Type Confusion Detection

Recently type confusion vulnerabilities received significant attention as an alternative memory corruption mechanism which is not covered well by static analysis and run-time checkers. CaVer [12] was designed as an efficient system to dynamically track type information and to perform type validation at potentially vulnerable cast locations. It uses the metadata system tracking in LLVM for heap objects, but reverts to red-black trees for stack and global allocations. As such, it requires additional operations during metadata retrieval to identify the type of the pointer. By using METAlloc, CaVer gains access to uniform pointer handling and low overhead irrelevant of the memory usage pattern. This is especially beneficial when considering the excessive overhead reported in CaVer for Firefox, which was attributed to its use of stack variables.

3.4 Dangling Pointer Detection

Use-after-free vulnerabilities represent the most prominent attack vectors in today's browser landscape [11]. While a lot of effort is invested to detect these vulnerabilities via static analysis and software testing, they typically manifest in highly specialized contexts, making them hard to detect and to fix preemptively. As such, a couple of systems have been suggested recently to mitigate the underlying reason for the vulnerabilities, dangling pointers [11, 15]. These systems rely on tracking heap allocations and their connectivity at run time. When an object is freed, the systems identify whether there are any pointers still pointing into the object being released. These pointers are then set to a benign value of NULL to mitigate potential memory dereferences using them. Systems for tracking dangling pointers share an underlying design based on three core data structures. The first is the object map, which identifies heap objects based on any pointer into the object itself (at any offset). This is equivalent to object metadata tracking. DangNull [11] uses red-black trees to track heap allocations, but as discussed in section 1, this scheme is susceptible to heavy and unpredictable overhead. FreeSentry [15] uses a label based system, which is equivalent to the fixed compression ratio metadata shadowing. This scheme offers fast fixed-time metadata retrieval, but incurs significant allocation-time and memory overhead. In contrast, METAlloc combines low allocation- and lookup overhead with efficient memory usage.

4. EVALUATION

To measure the performance impact of metadata tracking, we instrumented all the C and C++ SPEC2006 [9] benchmarks to observe the overhead it introduces. As a baseline, we compiled the applications with SafeStack enabled since it is advertised as a viable replacement for stack canaries, even showing lower overhead on some benchmarks [10]. We also use tcmalloc as the heap allocator for our baseline, as it can have very different run-time performance compared to the system allocator. This decision is also motivated by the fact that we observed a 10% improvement of execution times when using tcmalloc on SPEC2006 (geometric mean). This improvement increases to 17% when considering only the C++ benchmarks. For each benchmark, we used the median run time over 16 runs on a Xeon E5-2630 running CentOS Linux 7.2 64-bits.

Figure 3 shows the overhead introduced with the different configurations of METAlloc. We evaluated creating and initializing both 1 and 8-bytes of metadata for all objects. These setups correspond to different instrumentation types, like write integrity tracking or type hashes. The overhead numbers are very low, with the maximum being around 20% for perlbench and the geometric mean being 3.6% for one byte of metadata and 3.7% for eight bytes. The results also show that metadata size has a limited impact on the overall performance, showing that the variable compression ratio can help deal with applications requiring complex metadata. While the measured overhead only includes the metadata creation and initialization, not the instrumentation itself, the latter can be tuned with careful design and is the topic of future work using METAlloc.

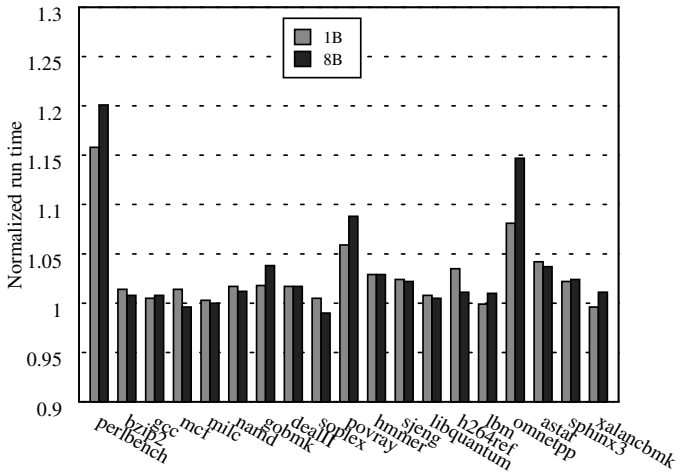

Figure 3: C/C++ SPEC2006 overhead with different configurations of METAlloc. 1(8)B represents the configuration with 1(8)-byte metadata entries.

5. CONCLUSION

In this paper, we presented METAlloc, a new memory metadata management scheme for software security hardening solutions. Our design is both comprehensive—given that it can handle whole-memory object metadata in a uniform and transparent way—and efficient—given that it yields a run-time performance overhead of just 3.6% in practice for (de)allocations. We believe METAlloc can bring many instrumentation solutions within reach for adoption in practice, allowing, for example, many vulnerability mitigation techniques to improve software security in an efficient and backward compatible fashion.

Acknowledgment This work is supported by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI "Dowsing", and by the European Commission through the project SHARCS under Grant Agreement No. 644571.

6. REFERENCES

[1] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[2] 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 Symposium*, pages 51–66, 2009.

[3] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[4] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171. ACM, 2006.

[5] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913. ACM, 2015.

[6] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009.

[7] I. Haller, A. Slowinska, and H. Bos. Mempick: High-level data structure detection in c/c++ binaries. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 32–41. IEEE, 2013.

[8] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144. ACM, 2012.

[9] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[10] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[11] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Internet Society Symposium on Network and Distributed Systems Security*, 2015.

[12] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 81–96. USENIX Association, 2015.

[13] J. Rafkind, A. Wick, J. Regehr, and M. Flatt. Precise garbage collection for c. In *Proceedings of the 2009 international symposium on Memory management*, pages 39–48. ACM, 2009.

[14] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.

[15] Y. Younan. Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Internet Society Symposium on Network and Distributed Systems Security*, 2015.