Fast and Generic Metadata Management with Mid-Fat Pointers

Taddeus Kroes[*]
t.kroes@vu.nl

Koen Koning[*]
koen.koning@vu.nl

Cristiano Giuffrida
giuffrida@cs.vu.nl

Herbert Bos
herbertb@cs.vu.nl

Erik van der Kouwe
vdkouwe@cs.vu.nl

Vrije Universiteit Amsterdam

ABSTRACT

Object metadata management schemes are a fundamental building block in many modern defenses and significantly affect the overall run-time overhead of a software hardening solution. To support fast metadata lookup, many metadata management schemes embed metadata tags directly inside pointers. However, existing schemes using such tagged pointers either provide poor compatibility or restrict the generality of the solution.

In this paper, we propose *mid-fat pointers* to implement fast and generic metadata management while retaining most of the compatibility benefits of existing schemes. The key idea is to use spare bits in a regular 64-bit pointer to encode arbitrary metadata and piggyback on software fault isolation (SFI) already employed by many modern defenses to efficiently decode regular pointers at memory access time. Our experimental results demonstrate that we cut overhead in half compared to a defense running on top of SFI, more than compensating for SFI overhead. Moreover, we demonstrate good compatibility, which may be further improved by static analysis.

1. INTRODUCTION

The research community has produced many defenses against common types of vulnerabilities in legacy code, and some of them are even available out-of-the-box in widely used compilers. However, such defenses are rarely deployed in production due to their prohibitive overhead. AddressSanitizer [21], for example, can detect a number of memory safety violations that threaten security, but incurs a performance overhead of 73%. We present a system that allows application developers to compose arbitrary combinations targeted defenses while achieving lower overhead than traditional frameworks.

In the literature, we observe two emerging trends. First, there is an increasing need for generic and composable defense frameworks, which can combine multiple defenses but disable others to tune the performance-security trade-off according to the deployment requirements [23]. Second, we need fast pointer-to-object metadata lookup schemes to empower such frameworks [12]. A promising strategy to design such schemes efficiently is to embed tags inside pointers.

Two tagged pointer schemes dominate the literature. Fat pointers change the size and representation of pointers to support arbitrary metadata tags [4, 14, 19]. This solution can support generic defenses with no restrictions, but it is plagued by compatibility problems since pointers radically change their format [6, 7]. Low-fat pointers carefully place objects in the memory address space to implement the other extreme: no changes in pointer size and representation, and a small metadata tag implicitly embedded in the most significant bytes of the address [6, 7]. While this solution clearly improves compatibility, it also greatly restricts the metadata tags that can be supported to a specific defense such as spatial safety [6,7]. Orthogonal and less efficient metadata management schemes become necessary to defend against other classes of vulnerabilities, such as dangling pointers [16,22,25] and type confusion [11, 17].

In this paper, we present *mid-fat pointers*, an approach to support targeted but composable metadata-based software defenses. The key idea is to strike a balance between traditional fat pointers and low-fat pointers by preserving the pointer size but changing its representation with embedded metadata. We show our design can support arbitrary metadata for generic defenses, while largely retaining the compatibility benefits of low-fat pointers. This is done by piggybacking on software fault isolation (SFI) [24] to decode the pointers efficiently, a technique already required by many modern defenses to protect sensitive data or metadata such as shadow stacks [9, 20]. In such a scenario, its use by mid-fat pointers incurs no additional overhead. Moreover, as we show in Section 6, the performance benefits from using pointers for this purpose more than compensates for the performance overhead incurred by SFI.

With mid-fat pointers, each pointer can embed the location of the metadata for the pointed memory object. Compared to recent generic metadata management schemes [12], we improve performance because we need fewer lookups as the metadata location is cached in the pointer and security because an out-of-bounds pointer is still associated with the same metadata. Moreover, our scheme does not require range lookups because the metadata lookup is performed at

[*]Equal contribution joint first authors.

EuroSec'17, April 23 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4935-2/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3065913.3065920

allocation time. This allows our scheme to be used as a caching layer on top of a wide variety of metadata management schemes. As such, we believe mid-fat pointers are a good basis to combine targeted defenses in arbitrary ways and achieve a practical performance-security trade-off suitable for real-world application deployments.

Contributions.

- We present mid-fat pointers, a new approach to support composable metadata-based software defenses.
- We evaluate mid-fat pointers and show that they incur low overhead in looking up and protecting metadata for memory objects.
- We show that it is possible to change the pointer representation with minimal impact on compatibility, even with the use of uninstrumented libraries.

2. THREAT MODEL

Since mid-fat pointers provide a general defense framework, the threat model depends on the particular defenses deployed. With regards to these defenses, we only assume they require an SFI baseline. As a result, the metadata is inherently protected against both arbitrary memory reads and writes, but we make no guarantees on the confidentiality or integrity of the application data. It is up to the developers to specify which threats the defense can handle and also deploy defenses for all the relevant threats.

3. OVERVIEW

Mid-fat pointers provide a framework to deploy software defenses against various classes of vulnerabilities. This section introduces mid-fat pointers from the perspective of *application developers*, who apply defenses to their software, and that of *defense developers*, who build security solutions on top of our framework.

3.1 Application developers

To use our framework, application developers only have to change their build system configuration. In particular, they need to use the Clang/LLVM compiler [15] with additional arguments to enable link-time optimizations, run our instrumentation passes (including one or more defense passes), and link to a modified memory allocation library with predetermined hooks. Furthermore, they need to prelink all shared libraries on which the binary depends. The developer does not need to modify the source code—unless required by the particular defenses considered.

3.2 Defense developers

To use our framework, defense developers add additional components to implement security solutions that delegate metadata management to mid-fat pointers. Figure 1 depicts the end-to-end workflow. Defense developers typically write a compiler pass to instrument particular operations that are potentially unsafe (for example array accesses, pointer arithmetic, pointer propagation, or type casts) as well as a static library to add code that hooks into both their own instrumentation and hooks offered by our framework. Our framework tracks pointers and offers a mechanism to efficiently look up defense-specific metadata based on a pointer value.

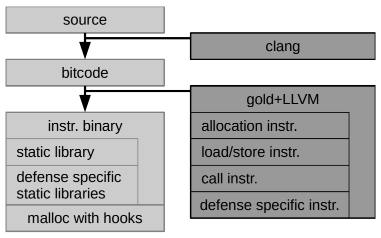

Figure 1: Overview of our framework

Allocation and deallocation hooks in the memory management library allow the defense to initialize and, if needed, clean up metadata for each heap object. Our allocation-time instrumentation adds tracking bits in each pointer to aid metadata lookup, while our load/store and call instrumentations prevent these bits from interfering with normal usage and simultaneously protect the integrity of the metadata from attackers.

4. DESIGN

In this section, we describe the design of mid-fat pointers. First, we discuss how we use an alternative pointer encoding to look up metadata in a way that improves both performance and security, while retaining compatibility. Next, we discuss how we instrument allocations. Finally, we consider how we manage metadata for memory objects.

4.1 Pointer encoding

One core insight for our framework is that, given the presence of SFI to protect metadata from attackers, we can use the unused bits in pointers without an additional performance impact. In particular, whenever the program allocates memory, we store a pointer to the metadata for that chunk of memory in the high bits of the pointer. This *mid-fat pointer* is then propagated automatically to any derived pointer. This design has three benefits: (1) it requires a full metadata lookup only once at allocation time, caching the more frequent lookups needed for defenses and increasing efficiency; (2) since we know the base address of newly allocated objects, we do not need range queries (normally required to associate an interior pointer into the middle of the object with object metadata), making allocation-time metadata initialization more efficient (as described in Section 5.1); and (3) pointers that go out-of-bounds still point to the metadata for the original object, increasing security in cases where an attacker can use pointer arithmetic to make a pointer point to a different object. In this section, we describe how we encode the location of the metadata into pointers and how we can ensure correctness even in the presence of uninstrumented libraries.

By default, we use the 32 lower bits of every pointer to encode where the data is (*data pointer*), and the 32 higher bits to encode where the metadata is (*metadata pointer*). Figure 2 shows how we decode the data pointer by masking out the metadata pointer bits, which is effectively the same operation used by SFI [24] to protect a part of the address space. While efficient, the downside is that we can use only 32-bit addresses out of the 48 bits implemented on x86-64, which restricts the address space to 4 GB. Note, however, that we could use fewer bits for the metadata pointer by assuming that every object is aligned on 8-byte boundaries.

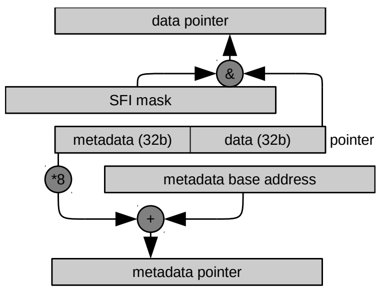

Figure 2: Encoding and decoding of pointers

We perform the encoding and decoding operations using the instrumentations shown in Figure 1: the allocation instrumentation encodes pointers, the load/store and call instrumentations decode data pointers, while defense-specific instrumentations decode metadata pointers. We now provide more details on these instrumentations.

4.2 Allocation instrumentation

We need to ensure that pointers returned to the application are properly encoded whenever it allocates memory. Unfortunately, we cannot simply replace `malloc()` with a version that performs this encoding, as even uninstrumented libraries would receive encoded pointers and any dereference would cause a segmentation fault. Instead, we use a compiler pass that searches for calls to memory allocation functions and adds a call to a hook in our static libraries. The hook looks up metadata for the allocated pointer (see Section 5.1) and adds the metadata pointer in the uppermost bits of the returned pointer as shown in Figure 2. Note that the implementation is inherently thread-safe, as newly allocated pointers are returned to the application only after being translated to a mid-fat pointer.

Load/store instrumentation.

As the program can only deference encoded pointers safely after decoding, our framework includes a compiler pass that identifies load and store operations and adds the data pointer decoder to each instance. Doing so requires only a single masking operation. We effectively implement SFI [24], shielding all memory past the first 4 GB from attackers even if they can perform arbitrary reads or writes. Thus, we can safely store the metadata above the 4 GB boundary.

Call instrumentation.

While instrumented code can safely dereference encoded pointers due to automatic masking, protected applications may well use unprotected libraries and we must not pass encoded pointers to them to avoid segmentation faults. We therefore instrument calls to external functions by decoding any pointer-type parameters. Note that we can identify which functions are imported from libraries because our pass is part of LLVM's link-time optimizations. For indirect calls our instrumentation might not know if the current value of a function pointer is to an external call or not. Instead, when a function is address-taken we provide a replacement function that calls the original function with masked arguments. It should be noted that, while we mask the parameters passed to uninstrumented library functions, there is no need for specific handling of returned pointers. A regular pointer can be safely used with our decoding scheme and our code detects that it does not point to metadata. In cases where the loss of the metadata is problematic, we could implement a hook to look up the metadata at return time and encode it in the returned pointer.

Pointer comparison and cast instrumentation.

Besides dereferencing operations, compatibility issues arise when comparing pointers, and in integer operations on pointers. The presence of a metadata pointer in the high bits may influence the outcome of these operations, changing the semantics of the program. For example, aggressive LLVM optimizations sometimes generate code that uses a pointer value as an offset after an exclusive-or (XOR) operation. Nonzero high bits will result in an incorrect offset, since the LLVM offset calculation does not account for metadata bits. We instrument these cases with the same masking operation we use for dereferencing instructions. To do this correctly, pointers must be distinguishable from integer values. This is a problem when dealing with union types, which can contain a pointer or an integer at the same location in memory. When such a value is loaded from memory, it is impossible to say with certainty whether it will be used as a pointer or as an integer. We currently do not mask these values. In future work, we could use static analysis to determine which pointers could be used in integer arithmetic and exclude them from our tagging scheme and SFI.

Defense-specific instrumentation.

Most defenses include instrumentation passes that instrument those operations that are either a security risk requiring a check or require tracking for later use. These passes can build upon our pointer decoding functionality to look up metadata. An encoded pointer yields a pointer to the metadata for the object, while regular pointers (derived from uninstrumented libraries) are easy to recognize as the metadata pointer is `NULL`. The size of a metadata entry is the sum of the (compile-time fixed) size requested by the defense, which means they can all share the same data structures by using an offset into the entry. As our framework does not interpret the metadata, defense developers are free to define them in any way they see fit.

4.3 Metadata management

The main feature provided by our framework is linking every *memory object* to a piece of *metadata*. A memory object is a chunk of memory allocated as one unit. The metadata consists of a fixed number of bytes, set at compile time, for use by the defense to which it belongs. Although the size of the metadata for each defense is set at compile time, it is possible to store a pointer in the metadata to build arbitrary-sized data structures.

Traditional metadata management schemes provide a simple API: they provide a function to look up metadata given an object pointer, and update their own data structures whenever an object is (de)allocated. There are various possible designs for the underlying data structures, including trees, hash tables, and shadow memory. The efficiency of these designs depends on the size of the metadata, the number and size of objects, the locality of memory accesses, the availability of memory, etc. Moreover, some systems allow range queries to look up interior pointers while others can only look up the base address of the memory object. We do not enforce the use of any one particular design but rather allow developers to pick the design most suitable for their purposes. We build on top of this simple API to look up metadata at allocation time to reduce the number of potentially expensive lookups and avoid the need for range queries. Our only additional requirement is that the metadata allocator be modified to allocate metadata in the SFI-protected memory area to ensure that it is out of reach for attackers. In practice, this is simply a matter of replacing memory allocation calls with calls to a library that allocates memory on top of an `mmap`'ed memory area.

5. IMPLEMENTATION

We now describe the end-to-end metadata management in our mid-fat pointers prototype, as well as how to shrink the address space to reuse pointer bits for our own purposes.

5.1 Metadata management

As mentioned earlier, mid-fat pointers may use any underlying metadata management scheme to store metadata structures. Our current framework prototype relies on METAlloc [12], which uses a memory shadowing scheme with a variable compression ratio to manage metadata. METAlloc itself relies on tcmalloc [8] memory allocator to ensure that all memory objects within a 4 KB memory page share the same alignment. It locates metadata using a two-step lookup. It first consults the *metapagetable*, which stores the alignment for each memory page as well as a pointer to an array of the metadata entries for all memory objects in that page. The second step is to divide the offset of the pointer in the memory page by the compression ratio (derived from the alignment) to determine which entry in the array should be used. As an object may be larger than its alignment, multiple entries in the metadata array may correspond to the same object. The main difference between our metadata management and METAlloc's is that we do not need range lookups, because we know the base address of the memory object when we look it up (see Section 4.1 for details). This means we can achieve better allocation performance by initializing only the first entry.

5.2 Shrinking the address space

In order to get enough bits in all application pointers for the metadata pointer, we have written a tool that reduces the address space of a program to any number of bits. On Linux, user address spaces are 47 bits and are very sparse: the program itself is often near the bottom of the address space, whereas the stack is near the very top. Allocations made using `mmap` are in between, slightly below the stack. Our tool consists of a post-build script and a static library.

The post-build script *prelinks* the binary, loader, and any dynamic library used by the program. It scans all these dependencies and constructs an address space where all these libraries fit below the new limit.

The static library hooks into the `preinit_array` to run as early as possible after starting the process. It looks up the existing mappings for the process and `mmaps` any free memory above the reduced address space limit to prevent arbitrary allocations containing more bits in their pointers than allowed. Next, we move both the thread local storage (TLS) and stack to the new reduced address space by creating a copy, updating all pointers to point to these new copies, and then set `FS_BASE` and `rsp` respectively. Currently we unmap the old stack, but have to keep the old TLS mapping around, as there are still pointers to it internally in `glibc`. This does not pose any issue because the application itself never sees such pointers. Any dynamic libraries loaded while the program runs (for example using `dlopen`) automatically end up in the shrunken address space as the remainder is reserved.

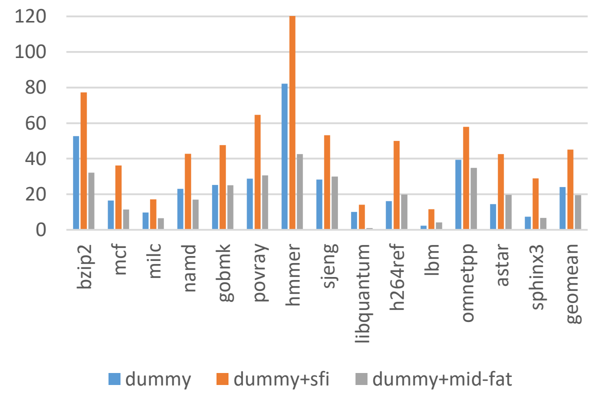

Figure 3: Performance overhead (%) on SPEC CPU2006

6. EVALUATION

To evaluate how well our framework performs, we built a system on top of it that retrieves the metadata for every object that the instrumented program writes to and verifies that it equals zero. We will refer to this mock defense as "dummy". This is similar to the work performed by WIT [2], but without static analysis and optimizations, or a write-only bounds checker. Such a workload is representative for a metadata management system, heavier than systems that instrument only specific operations (such as pointer writes or typecasts) but lighter than systems that also instrument reads (full bounds checking). We used this dummy defense to instrument the SPEC CPU2006 benchmarking suite [13] and ran the benchmarks on Intel Xeon E5-2630 machines with 16 cores at 2.40 GHz and 64 GB of memory, running the 64-bit CentOS 7.2.1511 Linux distribution.

To provide some context about our performance, we compare four configurations: the baseline does not use SFI, is compiled using LLVM with link-time optimizations (which implies the highest level of optimization), and uses the tcmalloc [8] allocator; the dummy configuration additionally has our dummy defense applied using traditional METAlloc [12]; the dummy+sfi configuration uses the defense, combined with SFI for reads and writes; and the dummy+sfi+mid-fat configuration applies our dummy defense using mid-fat pointers on top of METAlloc as a storage layer rather than using METAlloc directly.

We were able to run 14 out of the 19 SPEC CPU2006 C and C++ benchmarks, the remaining ones crashing due to invalid pointer dereferences. We investigated the problem with the Perlbench benchmark and found that it uses the same data structure for pointers and integer values. A pointer is loaded from this data structure as an integer, and used without being masked. Section 4.2 describes this problem and a possible solution in more detail.

Figure 3 shows the run-time overhead compared to the

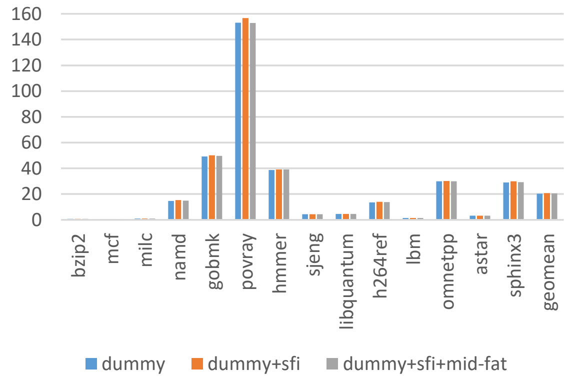

Figure 4: Memory overhead (%) on SPEC CPU2006

baseline for the three other configurations. Please note that the results are not comparable to those presented for METAlloc in [12], which does not use a dummy defense but rather instruments only allocations. It shows that our system incurs only 19.4% performance overhead overall on this subset of SPEC CPU2006. Traditional METAlloc on top of SFI incurs 45.1% overhead, which means that our pointer tagging scheme more than cuts overhead in half. METAlloc without SFI incurs 19.5%, which means that the performance gain from pointer caching is even larger than the performance loss of having to use SFI, resulting in a net speed-up. It should be noted however, that different defenses might gain more or less from using mid-fat pointers depending on the number of lookups that can be cached in pointers.

Figure 4 shows the memory overhead compared to the baseline. Overall memory usage on this subset of SPEC CPU2006 is virtually identical between the configurations, namely 20.4% for dummy, 20.7% for SFI, and 20.5% for mid-fat pointers. This shows that our approach does not incur additional memory overhead compared to the metadata storage layer it builds on.

7. LIMITATIONS

The main limitation of our system is the fact that it restricts the virtual address space to 32 bits, corresponding with 4 GB of addressable data. We believe this should be sufficient for most applications, many of which also work on 32-bit platforms, where they typically only have 2 to 3 GB available (operating systems usually reserve the remainder of the 4 GB address space for the kernel). Even Linux, in fact, includes the x32 ABI, explicitly designed to run x86-64 applications with 32-bit address spaces to reduce the memory footprint and improve performance [1]. A consequence of reducing the address space is also reducing the entropy available for address space layout randomization (ASLR). In principle, this should not be an issue if the proper defenses are deployed to deny the attacker access to read/write primitives capable of brute-forcing ASLR. Moreover, recent work [5, 9, 10, 20] has already shown that even 48-bit ASLR is also susceptible to derandomization attacks. We also note that all the non-fat pointer schemes including low-fat pointers inherently share similar limitations when adapted to support generic metadata. In the most generic case, two 32-bit pointers need to be encoded in a 64-bit pointer value to access both data and metadata.

Our framework requires source code to be available because it works at the LLVM bitcode level rather than on the final binary. This should not make much of a difference in practice as many defenses require source-level information to be able to add effective protections. Moreover, working on bitcode rather than binaries makes the framework more portable to other CPU architectures targeted by LLVM.

Our prototype only handles heap objects, not objects allocated on the stack or in global memory. This is merely an implementation limitation as the design to handle these cases in METAlloc [12] applies to our system as well. However, tracking stack objects will incur additional overhead.

Our prototype implementation does only limited static analysis on integer values that may be pointers (see Section 4.2), causing 5 SPEC benchmarks to fail. We plan to address this issue in future work.

Custom pointer-tagging schemes will conflict with mid-fat pointers, since our design assumes that the high 32 bits of a pointer are unused by the instrumented application.

There are cases where the program passes an instrumented pointer to a library that our analysis cannot catch. For example, through a global variable such as the environment pointer. Nonetheless, these cases could be handled with better static analysis, by instrumenting problematic libraries, or simple annotations.

We currently do not fully implement memory isolation in our proof of concept. That is, metadata allocations are performed in the lower 32-bit memory area and are thus not protected by masking operations. This is easily fixed by designating a memory area to metadata in the allocator and adding a constant offset at each metadata lookup. This arithmetic operation adds a minimal additional overhead.

8. RELATED WORK

Mid-fat pointers combine elements from two types of systems: those that encode information in pointers and those that use per-object metadata for defenses against software exploits. In this section, we consider both groups of systems.

Fat pointers.

The approach of changing the representation of pointers is known as “fat pointers” and traditionally refers to representations that increase the size of the pointer by adding fields such as the base and the size of the memory object pointed to, which allows for bounds checks. Examples include SafeC [4], CCured [19] and Cyclone [14]. However, this change is programmer-visible and tends to result in poor compatibility with existing code [18]. A solution is to use low-fat pointers [6, 7] instead, where the size of the pointer is not changed as some bits of the pointer are reused to encode metadata. However, this approach allows only very little data to be stored because 16 virtual address bits are not implemented in the MMU and must be masked on dereference. It also does not allow defenses to be composed, which means that it is only useful for simple bounds checks in practice.

Per-object metadata.

Approaches that store per-object metadata are more versatile. Such systems can be classified by the organization of their metadata: some use shadow memory, either with a fixed [3, 18, 21, 25] or with a variable compression ratio [11,12,22], while others use trees [16,17]. Our approach is most similar to METAlloc's [12] variable compression ratio scheme, but has efficient lookups because it uses pointer bits to cache the metadata pointer, more efficient allocations because it does not need range lookups, and it is more secure because it protects the metadata using SFI. It should be noted that the performance of many of these systems could be enhanced in contexts where SFI is needed by building them on top of mid-fat pointers, but we leave this for future work.

9. CONCLUSION

Metadata management is an important building block to build software security defenses. Existing schemes fall into one of the following three categories: those that do not store metadata information inside pointers (e.g., shadow memory-based schemes), those that implicitly store metadata information inside regular pointers without changing their size or representation (low-fat pointers), and those that explicitly store metadata information inside pointers by extending their size and representation (fat pointers). The last two tagged pointer schemes have the potential to support faster pointer-to-object metadata lookups, but also impose a trade-off between generality and compatibility.

In this paper, we presented an alternative design termed mid-fat pointers, which explicitly stores metadata information inside pointers (changing their representation) but without extending their size. Preliminary results show that mid-fat pointers offer good compatibility with existing applications and can support fast and generic metadata lookups on top of an SFI baseline. In increasingly common scenarios where SFI is necessary to preserve the security of a particular defense, we believe mid-fat pointers offer a competitive memory management scheme able to support arbitrary metadata structures with bounded performance guarantees.

To foster further research in the field, we have made the source code of our mid-fat pointers prototype available as open source at https://github.com/vusec/midfat.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This work was supported, in part, by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 64457, and the Netherlands Organisation for Scientific Research through the grant NWO 639.023.309 VICI "Dowsing".

11. REFERENCES

[1] The x32 system call abi. https://lwn.net/Articles/456731.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *S&P*, 2008.

[3] 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 SEC*, 2009.

[4] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[5] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory deduplication as an advanced exploitation vector. In *S&P*, 2016.

[6] G. J. Duck and R. H. Yap. Heap bounds protection with low fat pointers. In *CC*, 2016.

[7] G. J. Duck, R. H. Yap, and L. Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, 2017.

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

[9] E. Göktaş, R. Gawlik, B. Kollenda, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *USENIX SEC*, 2016.

[10] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017.

[11] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. In *CCS*, 2016.

[12] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos. METAlloc: Efficient and comprehensive metadata management for software security hardening. In *EuroSec*, 2016.

[13] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[14] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX ATC*, 2002.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

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

[17] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX SEC*, 2015.

[18] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI*, 2009.

[19] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *TOPLAS*, 2005.

[20] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *USENIX SEC*, 2016.

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[22] E. van der Kouwe, V. Nigade, and C. Giuffrida. DangSan: Efficient use-after-free detection. In *EuroSys*, 2017.

[23] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. High system-code security with low overhead. In *S&P*, 2015.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[25] Y. Younan. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.