MetaTableLite: An Efficient Metadata Management Scheme for Tagged-Pointer-Based Spatial Safety

***Abstract*—A tagged-pointer-based memory spatial safety protection system utilizes the unused bits in a pointer to store the boundary information of an object. This paper proposed a hybrid metadata management scheme, MetaTableLite, for tagged-pointer-based protections. We observed that objects of a large size only take a minority part of all the objects in a program. However, recording their boundary metadata with traditional pointer tags will incur large memory overheads. Based on this observation, we introduce a small supplementary table to maintain metadata for these few large objects. For small ones, MetaTableLite represents their boundaries with a 14-bit pointer tag, well utilizing the unused 16 bits in a conventional 64-bit pointer. MetaTableLite can achieve a 6% average memory overhead without alternating the conventional pointer representation.**

***Index Terms*—spatial memory safety, tagged pointer, metadata table**

I. Introduction

Memory spatial errors, i.e., buffer overflows, have been a well-known issue for computer security in the last decades. Such errors are still rife in modern C/C++ binaries and remain one of the root causes of exploitable vulnerabilities [1]–[3]. Various forms of attacks such as Return Oriented Programming (ROP) [4] and Data Oriented Programming (DOP) [5], [6] can be triggered with a memory spatial error.

A typical method to protect programs from memory spatial errors is to check the validity of every pointer used for memory access during runtime, which requires the system to record the boundaries of every memory object. Tagged-pointer-based schemes utilize the unused bits in a pointer for metadata storage. In such systems, the pointer tag encoding is a core part of the overall design. Intuitively, we can assign a unique ID for every memory object [7], [8] and index a metadata table with the ID. The metadata table can incur large memory overheads for small objects, because the bounds data may be as large as, or even larger than the object itself. Other systems introduce constraints for memory allocations to calculate object bounds from the pointer tag [9]–[11]. Usually, the constraints are put on object alignments and sizes, which can bring considerable overheads for large objects.

In this paper, we had an in-depth analysis of the pointer tag design in a tagged-pointer-based protection system. From the memory allocation traces, we observed that objects of very large size were rare. Based on this observation, we proposed a hybrid metadata management scheme. We use a small supplementary table to maintain metadata for large objects. For small objects, we introduce allocation constraints and use a floating-point fashioned tag to represent their bounds. By carefully selecting the large object threshold, we can effectively control the number of large objects, making the metadata table index less than 16 bits. We call this scheme MetaTableLite and implemented a prototype on LLVM 10.0. When setting the large object threshold to 1MB, it is able to achieve a 6% average overhead on virtual memory size (VmSize).

II. Related Works

Generally, tagged-pointer-based protection schemes can be classified into two categories. One uses the tag to index a metadata table, and the other uses the tag and the current address to calculate object bounds.

A. Indexing a Metadata Table

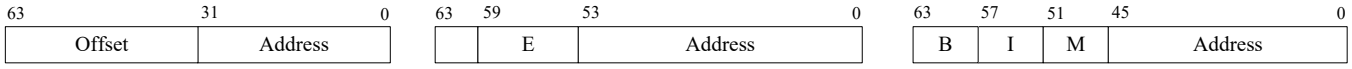
We can assign a unique ID for every object and use this ID to index a bounds metadata table. The necessary bits to encode object IDs are determined by the number of active objects. Unfortunately, in some programs, this number may be huge and cannot be represented with 16 unused bits. As a result, we must either find a way to compress the object ID or change the pointer representation to get more spare bits.

CUP [7] changes the pointer representation to spare more bits. CUP stores the base address and size of the object in the metadata table. The 32 higher bits of a pointer in CUP act as the object ID, and the 32 lower bits represent the offset from the current position to the base address of the object. CUP does not put constraints on the base addresses but limits the size of an individual object because its offset is limited to 32 bits.

AOS [8] utilizes Arm pointer authentication (PA) primitives [12] to compress the object ID. AOS uses a 16-bit PAC to index a hashed bounds table and concatenates PACs with partial memory addresses as tags for the hash table. When a collision happens, AOS dynamically enlarges the hash table, which will lead to a costly table resizing process.

B. Calculating Object Bounds

To restore object bounds with the current address and the tag, existing schemes tend to introduce extra constraints to the usable address space or object allocations. For example, Deltapointer [9] uses the distance from the current position to the upper bound as the pointer tag. It adjusts the usable address space to 32 bits and stores the offset in the higher 32 bits of a pointer. Fig 1(a) shows the format of Deltapointer. It can only detect overflows because the pointer tag only represents the offset of one direction.



(a) Deltapointer (b) M-Machine (c) Low-Fat Pointer

Fig. 1. Pointer formats in existing schemes

To calculate both upper and lower bounds from a pointer tag, a general method is to introduce alignment and size constraints to object allocations. M-Machine [10] requires objects to be 2^E aligned and 2^E sized. With the size constraint, M-Machine can encode the size of an object with only 6 bits. And with the alignment constraint, M-Machine is able to calculate the base address by setting the lowest E bits to zero. Fig 1(b) shows the pointer format of M-Machine. It adopts a 54-bit address space and uses the highest 4 bits for access permissions.

The M-Machine format is elegantly simple yet also incurs considerable memory fragmentations. In the worst case, its size constraint can introduce a 50% overhead to a single object. Low-Fat Pointer [11] adopts a more fragmentation-friendly encoding format named BIMA. BIMA represents the object size in a floating-point fashion with three subfields B, M, and I. It views an object as a combination of several consecutive 2^B-byte blocks and requires it to align on a 2^B-byte boundary. The minimum field, I, specifies the base address, and the maximum field, M, specifies the bound address. The BIMA format takes 18 bits for tag encoding as shown in Fig 1(c), decreasing the usable address space to 46 bits.

III. MetaTableLite

A. Observations on Allocation Traces

We extracted memory allocation traces from several SPEC CPU 2017 [13] benchmarks with SystemTap [14]. SystemTap catches all the memory allocations and records the sizes of heap objects. It does not catch the allocations of stack objects. We chose 12 pure C/C++ programs because C/C++ programs are more vulnerable to memory spatial violations and most defenses aim at them.

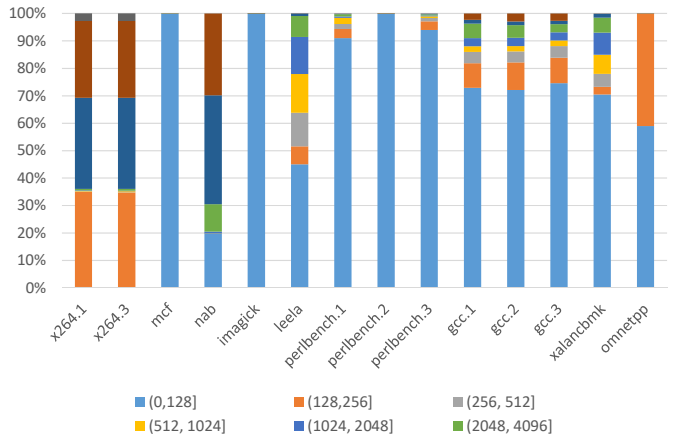


Fig. 2. Distribution of object sizes

From the allocation traces, we found that in some programs, active objects can be as few as less than 10. While in other programs, they can be as many as more than 2 million. Encoding unique IDs for more than 2 million objects requires at least 22 bits. We can only spare 16 bits in a pointer without changing the conventional address space size, so we cannot simply assign an ID for each object.

We further investigated the distribution of object sizes. Fig 2 shows the results in benchmarks that have more than 1000 allocations. We notice that objects smaller than 256 bytes take up at least 20% in all of the benchmarks and take the overwhelming majority in many of the benchmarks. On the opposite hand, objects larger than 1MB are rare. The *mcf* is an extreme case: nearly all of its allocations are between 8 bytes and 16 bytes. And *imagick* has its allocations concentrate in (64B, 128B]. The *deepsjeng* is another outlier. It only allocates 4 objects, among which there is an object over 6GB.

Fig 2 shows that large objects are rare. If the number of such objects is low enough to encode with 16 bits, then it is possible to maintain a dedicated metadata table for them. Now the question is by what standard to classify an object as "large". We investigated the peak number of large objects under different thresholds, also excluding programs whose allocations are less than 1000. The results are shown in Table I.

TABLE I
LARGE OBJECT AMOUNTS UNDER DIFFERENT THRESHOLDS

	48KB	64KB	128KB	256KB	512KB	1MB
x264.1	230	228	198	188	94	94
x264.2	78	76	46	35	15	15
x264.3	230	228	198	188	94	94
mcf	5	5	5	5	5	5
nab	82	82	76	48	27	22
imagick	20	14	7	5	5	5
leela	7	7	6	6	6	4
perlbench.1	43	34	22	6	2	0
perlbench.2	11	11	10	9	9	9
perlbench.3	29	27	18	6	0	0
gcc.1	42025	264	74	37	25	23
gcc.2	43608	344	77	37	16	14
gcc.3	43473	327	76	38	19	13
xalancbmk	30	24	14	6	2	2
omnetpp	5	5	4	2	1	0

The peak number of large objects drops significantly when the threshold increases from 48KB to 64KB. When choosing 64KB as the threshold, the peak numbers are all below 500. If we maintain 16-byte metadata for each object, then the memory needed for metadata of 500 objects is less than 8 KB, which is negligible for SPEC CPU 2017 benchmarks. And considering the tag encoding, 9 bits are fairly enough to encode the index of a 500-entry table, making it possible to

store a table index without changing the conventional pointer representation.

We then simulated the overall memory overheads of the BIMA format [11] with the allocation trace. The baseline statistics were gained by adding up all the sizes in the trace. From the results in Fig 3, we can find that increasing the I field length effectively reduces memory overheads. As the I field increases from 3 bits to 6 bits, the average overhead drops from 22.1% to 3.0%.

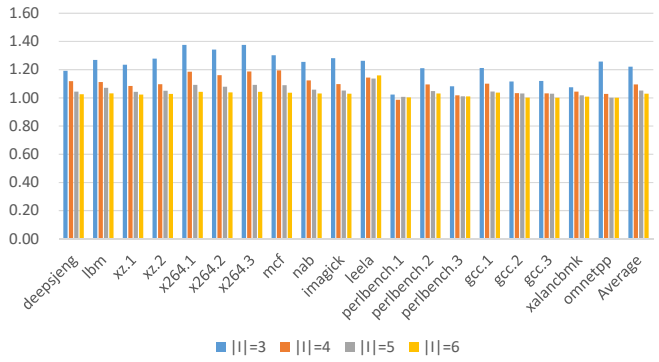


Fig. 3. Normalized peak VmSizes

B. Design Overview

Based on our observations, we propose a hybrid metadata management scheme. For small objects, we use the BIMA format to record their bounds with pointer tags; for large objects, we introduce a supplementary table to record their bounds. We call this scheme MetaTableLite. The formats of our pointer tag are shown in Fig 4.

0	B : 5	I : 4	M : 4		Address : 48

1	Index : 13		Address : 48

Fig. 4. Pointer formats of Extended Tagged Pointer

The most significant bit of a pointer is used to distinguish the two tag types. When it is 0, the pointer tag follows the BIMA format; otherwise, the pointer tag acts as the metadata table index. According to our previous evaluations, the B field needs to be at least 4 bits, granting a 64KB threshold to control the number of large objects. We assign 5 bits for the B field to grant more threshold choices. For the I field length, 4 bits and 5 bits are all possible choices. A 5-bit I field brings a lower overhead but takes up all the available bits. Out of extensibility considerations, we assign 4 bits for the I field and reserve 2 bits for later use.

We reserve two bits in our pointer encoding to serve as state bits. For example, they can state whether the pointer is out-of-bounds, or if it points to the one-past location. We may also use them to extend the metadata table, like introducing a type layout table. We leave this topic for future works.

Fig 5 represents the checking process. A bounds calculating unit (BCU) interprets the tag in BIMA format and calculates the upper and lower bounds. Because Low-Fat Pointer only uses simple arithmetic operations to restore object bounds, BCU can utilize the ALU or be extended from the address generation unit of the load/store unit. At the same time, the tag is also used as an index to retrieve the object bounds from the metadata table. The retrieval process can be costly, but according to previous works [8], [15], it can be effectively accelerated with a dedicated cache. Finally, the checking unit selects the bounds for pointer validation based on the most significant bit.

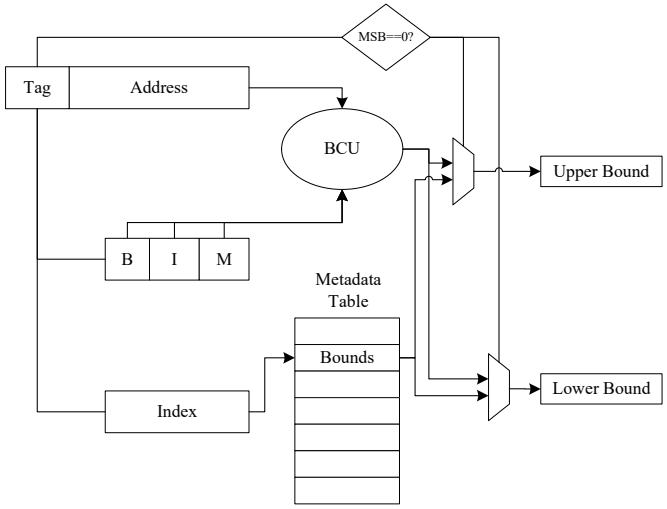


Fig. 5. Hardware overview

IV. Evaluation

A. LLVM Implementation

We implemented the MetaTableLite on LLVM 10.0. To generate pointer tags, we implemented wrapped allocation functions, *tagged_malloc()* and *tagged_free()*, to handle heap objects. For stack objects and global variables, we added an extra pass to instrument the LLVM Intermediate Representation (IR). The *alloca* IR instruction generates stack objects, so we calculate the tag for stack objects during the compilation and insert instructions after every *alloca* to replace the pointer with the tagged one. As for global variables, we adopted Deltapointer's method, introducing a dedicated constructor function to handle them [9].

In our prototype, the threshold for large objects ranges from 1MB to 256KB. According to our previous investigations, large objects will not exceed 500 under these thresholds. In *gcc*, we found stack objects with a variable length, whose size cannot be known during the compilation. Such objects are only found in *gcc* and are rare. We skipped this benchmark for the moment, but such objects can be handled by inserting instructions to calculate their sizes during runtime and using a dedicated function to update the metadata table. We also checked all of the compiler-visible global variables and stack

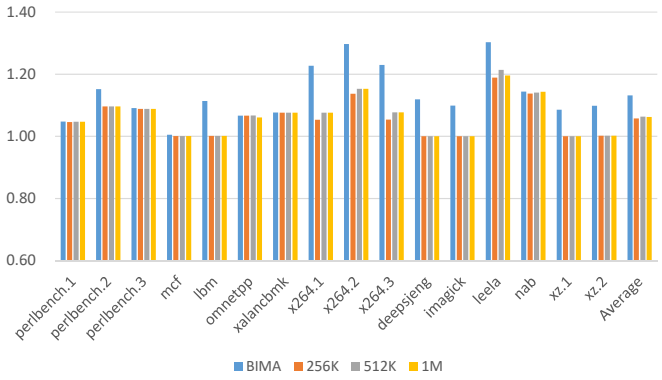


Fig. 6. Normalized VmSize of LLVM implementation

objects and only found one large global variable in *deepsjeng*, which was handled in the constructor function. For stack objects, all of their sizes are beneath the threshold. According to the investigation in CUP [7], the number of stack objects can be very large, but our MetaTableLite can maintain all of their metadata with the pointer tag. MetaTableLite does not rely on the metadata table to handle stack objects and avoids adding massive memory accesses because of metadata retrieval.

We compiled the SPEC CPU 2017 benchmarks with our modified LLVM and measured the VmSize of each benchmark. Here we do not measure the RSS result because we want to analyze the influence of tagged pointers on the whole working set, excluding the influence of the OS memory scheduling. The baseline VmSizes come from original SPEC benchmarks. The "BIMA" results in Fig 6 are measured by setting the I field to 4 bits and applying no threshold. From Fig 6 we can find that this hybrid management scheme can effectively reduce the program VmSize, and adjusting the threshold only brings minor changes to the overheads. MetaTableLite can achieve a 6% average overhead, bringing a reduction of around 7% in contrast to Low-Fat Pointer. For some of the benchmarks, MetaTableLite shows a significant overhead reduction. It can reduce the overhead of *x264* by 16-18%. It also shows an improvement of around 10% on *leela*.

V. Conclusions

This paper had an in-depth analysis of the pointer tag design in tagged-pointer-based memory protection schemes. We observed that the majority of objects had a small size and that objects of very large sizes are rare. Based on this observation, we proposed a hybrid bounds metadata management scheme that records the bounds of small objects with pointer tags and maintains the bounds of large objects with a supplementary table. Because of the rarity of large objects, this hybrid scheme only needs to maintain a small bounds table while effectively reduces the memory overheads. We call this hybrid scheme MetaTableLite. We implemented a prototype on LLVM 10.0. When setting the large object threshold to 1MB, the objects that should be recorded in the metadata table can drop from several million to less than 500. At the same time, it is able to achieve a 6% average overhead for virtual memory size (VmSize) on SPEC CPU 2017. In contrast to a pure floating-point encoding scheme, MetaTableLite is able to reduce memory overheads by 7% on average and 16-18% in the best case.

References

[1] V. Van der Veen, L. Cavallaro, H. Bos *et al.*, "Memory errors: The past, the present, and the future," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2012, pp. 86–106.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62.

[3] Y. Younan, "25 years of vulnerabilities: 1988–2012," *Sourcefire Vulnerability Research Team*, 2013.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *USENIX Security Symposium*, vol. 5, 2005.

[6] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 969–986.

[7] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 381–392.

[8] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1153–1166.

[9] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 22:1–22:14.

[10] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: Association for Computing Machinery, 1994, p. 319–327.

[11] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 721–732.

[12] ARM, "Arm cortex-a series - programmer's guide for armv8-a," 2015.

[13] "Spec cpu 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017

[14] F. C. EIgler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen, "Architecture of systemtap: a linux trace/probe tool," 2005.

[15] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 762–775.