Safer Heaps with Practical Architectural Security Primitives

William Robertson[1] and Manuel Egele[2]

[1]*Northeastern University*
[2]*Boston University*

Introduction

After working in security for a while, it sometimes seems inevitable to conclude that the more things change, the more they stay the same. In this context, given the importance of a secure cyberspace for modern society, this constancy is decidedly not a good thing. Memory corruption vulnerabilities continue to plague software, and the security community continues to be unable to completely detect and remove or neutralize these vulnerabilities, to predict new classes of vulnerabilities, or otherwise make strong guarantees with respect to memory safety for general software.[1] And today, heap vulnerabilities comprise a large fraction of memory corruption vulnerabilities. Heap vulnerabilities are especially pernicious because detecting them requires enforcing temporal memory safety, a strictly harder task than enforcing spatial memory safety.

It isn't all doom and gloom. Years of research on software mitigations have yielded considerable progress in heap vulnerability detection and hardening code against their exploitation. Modern allocators can enforce heap integrity with cookies, enforce correctness constraints on heap metadata like free list pointers, or track pointers to freed memory to avoid double-free exploits [5, 1]. Hardening the heap carries significant promise for mitigating real-world exploitation of software written in unsafe languages.

At the same time, it is unclear how much further software-only approaches can be pushed. In practice, adopting stronger protections has to be weighed against the performance overhead perceived by users. Software vendors, in turn, have demonstrated a low tolerance for any security mechanism that qualitatively impacts user experience.

We argue that adopting architectural security primitives as the basis for allocator hardening research is the most promising path forward. An architectural security primitive (ASP), as opposed to a more general hardware security primitive, is part of the system's architecture and thus directly interacts with and is configured by the software executing on the system. This is in contrast to micro-architectural or device-level features that all contribute to the execution of software but are not directly exposed to it. ASPs hold significant promise for deterministic mitigation of temporal memory safety vulnerabilities without incurring intolerable performance overhead – one can have their cake and eat it too!

Intel MPK and MPKAlloc

As evidence of this, consider Intel Memory Protection Keys (MPK). MPK is a security extension to the Intel ISA that allows developers to partition an address space into distinct page-granularity domains tagged with one of 16 keys. MPK designates bits 59–62 of each page table entry (PTE) as that page's protection key.[2] MPK additionally adds a protection key rights register for user pages (PKRU) to each CPU thread. At every memory access, the

[1]To head off the obvious: yes, rewriting everything in a (mostly) memory-safe language would help matters greatly! But, let us also acknowledge that this is not economically feasible.

[2]Thus, MPK is a form of tagged memory, an architectural security approach that dates back to 1960s-era LISP machines and Burroughs mainframes.

MMU checks whether the corresponding PTE's protection key is present in the current thread's PKRU register – either reading or writing can be separately permitted. If so, the access is allowed and execution continues; otherwise, a hardware exception is raised. The PKRU can be updated using a special instruction (`wrpkru`), making domain switches relatively efficient. Also, since the MMU enforces access policies, domain checks are also efficient, incurring essentially zero overhead.

MPK has been used in numerous security applications since its introduction in the Skylake architecture in 2015. Despite the limited number of keys available and the "sharp edge" in that `wrpkru` is an unprivileged instruction, MPK is an excellent example of an ASP that can be used to mitigate temporal memory safety vulnerabilities. In fact, we demonstrated this capability with MPKAlloc in 2022 [2].

MPKAlloc builds on MPK to isolate allocator metadata from the rest of a program, effectively partitioning a program into a trusted allocator and an untrusted, potentially vulnerable, program. The key invariant MPKAlloc preserves is that any access made to allocator metadata must originate from a trusted domain, and the trusted domain only contains allocator code. By default, all CPU threads execute within an untrusted domain. Upon any allocator invocation – e.g., to allocate or free a heap chunk – MPKAlloc switches the thread to the privileged domain to enable metadata access. Once the requested allocator operation is performed, the CPU thread's rights are restored to the unprivileged domain.

wkr: We can probably fill out the rest of the word count with more MPKAlloc design and eval.

- Safety guarantees
- Low to no performance overhead
- Transparent to applications (minimizing developer burden)

Beyond MPKAlloc

ASPs represent a very promising path to mitigate memory safety vulnerabilities and constrain attackers. However, this path also presents several daunting challenges. Designing, implementing, and deploying hardware mitigations is substantially more capital intensive than analogous software mitigations. Thus, a failure of adoption or, even worse, the discovery of fundamental design flaws, represent a massive waste of capital investment and developer resources.[3] Moreover, unused hardware wastes precious power and area, and incurs support costs without any counterbalancing benefit.

MPKAlloc, and ASPs more broadly, are also not a panacea. While approaches like MPKAlloc severely constrain attackers by reducing an existing attack surface, history suggests that declaring victory against temporal memory safety vulnerabilities would be premature. For instance, MPKAlloc assumes that the allocator is correct and cannot be tricked into corrupting its own metadata. However, no such proof of these assumptions exists and thus one cannot preclude the possibility of that line of attack.

A major limitation of MPK-based defenses is the continuing limited availability of MPK in the wild. To date, MPK is still only present in certain models of Intel Xeon server class CPUs, which greatly limits the impact of MPKAlloc and its ilk. However, the limited availability argument does not apply to ASPs as a whole, and in particular to a similar ASP gaining popularity in the mobile space: Arm's Memory Tagging Extension (MTE).

First introduced in the Arm v8.5-A ISA, MTE is another implementation of the ever-popular tagged memory security architecture. MTE tags memory regions with "colors" similarly to how MPK tags pages with protection keys. However, MTE is significantly more powerful in that regions need not be page-sized. In addition, systems like HAKC have demonstrated that combining MTE with Arm Pointer Authentication (PAC) can greatly enlarge the space of tags [1].

However, the most exciting aspect of Arm MTE by far is twofold: *(i)* Google's commitment to adopt it as a platform security feature in Android, and *(ii)* the availability of both PAC and MTE in the Pixel 8, Google's current flagship mobile device. For the first time in the modern computing era, a practical and performant tagged architecture is now available for a major consumer market. This development opens up a world of possibilities for vulnerability mitigation, but one immediate and tantalizing prospect is to harden Android memory allocators using a similar approach to MPKAlloc. In fact, public information suggests that Google is pursuing exactly this path for PartitionAlloc, one of the allocators used by Chrome and also covered by MPKAlloc [3].

[3] The story of the ill-fated Intel Memory Protection Extensions (MPX) is instructive in this context [4].

With a robust and efficient tagged architecture in the form of Arm MTE now available, the future looks bright for ASP-based exploit mitigation. However, hardware vendors will only continue to invest in ASPs if they will be used; if nothing else, unused hardware is a support burden and wastes valuable power and area. Software vendors create a strong demand signal: adoption ultimately drives the perception of value of architectural security features, which incentivizes other hardware platforms to adopt similar functionality so as not to be on the wrong side of an important market differentiator.

Researchers also have an important role to play. The value of an ASP increases with its perceived capabilities. Thus, the more vulnerability classes researchers can mitigate with an ASP, the better incentivized hardware and software vendors will be to engage in a virtuous cycle of investment and innovation in next-generation architectural mitigations. Along similar lines, as we have already observed, minimizing developer burden is an important factor for adoption. Thus, the more that researchers can automate specification, design, and enforcement of ASP-based enforcement schemes, the better. Put simply, we sincerely hope that the research community in tandem with software vendors will help fulfill this important role and bring the world closer to that elusive goal of a world free from memory safety vulnerabilities.

References

[1] Lukas Bernhard, Michael Rodler, Thorsten Holz, and Lucas Davi. "xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel X86-64". In: *Proceedings of the IEEE European Symposium on Security and Privacy*. 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P). June 2022, pp. 502–519. DOI: 10.1109/EuroSP53844.2022.00038.

[2] William Blair, William Robertson, and Manuel Egele. "MPKAlloc: Efficient Heap Meta-data Integrity Through Hardware Memory Protection Keys". In: *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Lorenzo Cavallaro, Daniel Gruss, Giancarlo Pellegrino, and Giorgio Giacinto. Vol. 13358. Cham: Springer International Publishing, 2022, pp. 136–155. DOI: 10.1007/978-3-031-09484-2_8.

[3] Google Project Zero. *Project Zero: First Handset with MTE on the Market*. Project Zero. Nov. 3, 2023. URL: https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html.

[4] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. *Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches*. June 16, 2017. DOI: 10.48550/arXiv.1702.00719. arXiv: 1702.00719 [cs]. preprint.

[5] Brian Wickman, Hong Hu, Insu Yun, Daehee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. "Preventing {Use-After-Free} Attacks with Fast Forward Allocation". In: *Proceedings of the USENIX Security Symposium*. 30th USENIX Security Symposium (USENIX Security 21). 2021, pp. 2453–2470. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/wickman.