Should Your Database Systems Use Hardware-Assisted Memory Safety Extensions in Production?

Ilya Meignan--Masson†
Technical University of Munich
Germany

Martin Fink†
Technical University of Munich
Germany

Masanori Misono
Technical University of Munich
Germany

Dimitrios Stavrakakis
Technical University of Munich
Germany

Pramod Bhatotia
Technical University of Munich
Germany

Abstract

Database systems are predominantly developed in unsafe languages (e.g., C/C++) to meet the performance requirements through low-level memory management, yet this reliance renders them prone to systemic memory-safety issues that compromise reliability, consistency, security, and durability. Through an extensive bug analysis of prominent database systems, we show that these memory-safety issues persist in production environments despite advancements in database testing tools. While emerging hardware-assisted extensions, such as Arm's Memory Tagging Extension (MTE) and CHERI, offer a promising mitigation path for memory safety, their practical applicability and performance overhead within the specialized constraints of database systems remain largely unexplored.

In this paper, we evaluate these modern hardware extensions through the lens of what we define as the *database trilemma*: the fundamental trade-off between *safety*, *performance*, and *portability*, to determine their viability for production-grade database systems. We provide the *first* side-by-side comparison of MTE and CHERI across a database workload suite. Our bottom-up study spans microarchitecture, the compiler/runtime/OS stack, core data structures (ART, B+Tree, hash table, skip list, linked list, and queue), and full database systems (Redis, LevelDB, SQLite, MySQL, DuckDB, and LadyBugDB) to characterize their performance, safety guarantees, and ease of adoption.

We find that while hardware extensions can offer near-deterministic protection, they introduce non-uniform performance taxes: MTE provides high portability with modest overheads (≈10%), while CHERI delivers superior safety guarantees with higher performance penalties (20–60%) and significant porting effort. Our study equips database systems architects with an actionable guide towards building the next generation of reliable, secure databases using hardware-assisted safety mechanisms. [1]

†Both authors contributed equally.

[1]This manuscript is the extended version (with complete Appendix) of the paper accepted at SIGMOD'27 [92].

Authors' Contact Information: Ilya Meignan--Masson, Technical University of Munich, Germany; Martin Fink, Technical University of Munich, Germany; Masanori Misono, Technical University of Munich, Germany; Dimitrios Stavrakakis, Technical University of Munich, Germany; Pramod Bhatotia, Technical University of Munich, Germany.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/12-ART378
https://doi.org/10.1145/3856373

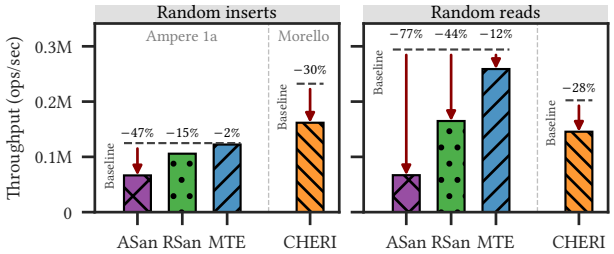


Figure 1: Overheads of ASan, RSan, MTE, and CHERI on LevelDB.

CCS Concepts

• **Information systems** → **Data structures**; **Database management system engines**; • **Software and its engineering** → **Memory management**; **Software safety**.

ACM Reference Format:
Ilya Meignan--Masson, Martin Fink, Masanori Misono, Dimitrios Stavrakakis, and Pramod Bhatotia. 2026. Should Your Database Systems Use Hardware-Assisted Memory Safety Extensions in Production?. *Proc. ACM Manag. Data* 4, 6 (SIGMOD), Article 378 (December 2026), 22 pages. https://doi.org/10.1145/3856373

Artifact Availability: The artifacts are available at https://github.com/TUM-DSE/memsafedb.

1 Introduction

Database systems are predominantly implemented in memory-unsafe languages (e.g., C/C++), to meet the performance requirements of data-intensive applications. Such languages provide developers with fine-grained control over memory management, enabling practitioners to design high-performance buffer managers, indexing structures, and query execution engines. However, this power comes at a significant cost: *the lack of inherent memory safety*.

Memory safety denotes the guarantee that memory accesses are well-defined and restricted to their intended regions; violations of this property [126, 133] are categorized as *spatial* or *temporal*. *Spatial errors* (e.g., overflows, out-of-bounds access) occur when a program accesses memory outside its intended boundaries, while *temporal errors* (e.g., use-after-free, double-free) refer to when a program accesses memory after its deallocation. In database systems, such violations constitute fundamental reliability, consistency, durability, and security issues.

The software industry has long grappled with memory safety [7,

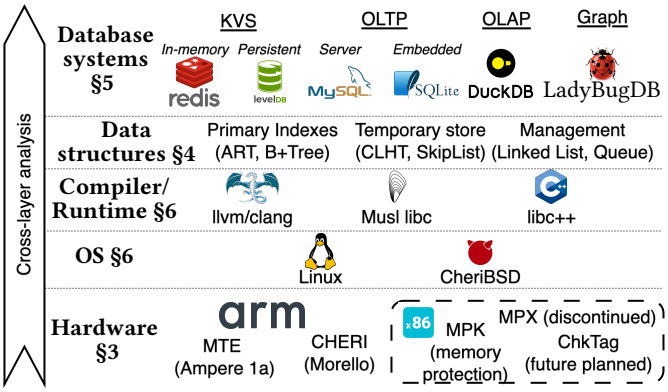


Figure 2: Analysis overview from hardware to full systems.

10, 61], with Microsoft and Google reporting that ≈70% of their reliability and security issues stem from memory safety bugs [8, 9, 112]. Database systems are no exception; our study across major open-source database systems (§2.1) reveals that memory-safety bugs remain a substantial share of all reported issues, and that even recently developed systems [31, 42] are not immune, confirming that memory safety remains a serious concern.

One option to address these issues is to rewrite existing systems in memory-safe languages (e.g., Rust), which offer performance while providing built-in safety guarantees. While such rewrites have proven successful, both for new projects [35] and for rewriting core components of production systems [65], they remain a substantial undertaking for systems comprising millions of highly optimized lines of code, where the engineering cost is often prohibitive.

As an alternative, database practitioners commonly resort to *exhaustive testing* and *software-based runtime checks*. While formal verification can prove safety, it fails to scale to the complexity of modern database systems. Meanwhile, *software-based runtime approaches* [54, 73, 85, 120, 121] are prohibitively expensive (up to 79% for lookups in LevelDB), as shown in Figure 1, and are, in general, not practical for production database systems.

To this end, hardware vendors have proposed *new hardware extensions* to provide memory safety with lower performance overheads. In particular, several propositions such as ARM's Memory Tagging Extension (MTE) [12], Capability Hardware Enhanced RISC Instructions (CHERI) [141], x86's upcoming ChkTag [33], Intel's deprecated Memory Protection Extensions (MPX) [13], and Oracle's SPARC Silicon Secured Memory [6] offer new primitives for memory safety. As shown in Figure 1, such extensions reduce overhead considerably compared to software-based approaches—yet they still impose non-trivial costs, and adopting them requires non-trivial code modifications, raising portability concerns.

Together, these constraints expose a *"Database Systems Trilemma"*: a fundamental conflict between *safety*, *performance*, and *portability*. In essence, developers must sacrifice performance for safety, or portability for a modern safe language, or, as is most common, sacrifice safety for the performance and control of unsafe languages.

While these technologies are increasingly adopted in mobile ecosystems [25, 51], they have been slow to reach database systems, mainly due to the lack of a systematic analysis and experimentation with these hardware extensions in this context. To this end, we aim to answer the central question: *Should your database system use hardware-assisted memory-safety extensions in production?*

To provide an answer, we conduct the first side-by-side, *cross-layer* comparison (Figure 2) of memory-safety hardware extensions tailored for database systems, spanning from microarchitecture through data-structures to full systems. Our study begins with a *microarchitectural analysis* of the underlying hardware primitives (§3). Next, we evaluate how these primitives affect the *"building blocks" of database systems* by porting and benchmarking a set of critical data structures (§4). We scale our analysis to *production-grade* database systems, detailing the engineering effort and performance trade-offs of retrofitting them with hardware safety (§5). Finally, we analyze the role of the operating system and compiler toolchains for hardware-assisted memory safety (§6).

To ground our analysis in practice, we distill our findings into actionable takeaways for database practitioners. At the microarchitectural level, we show that while hardware extensions effectively mask the cost of runtime checks, the overhead of managing metadata—such as CHERI's 128-bit capability loads or MTE's tag-on-store requirements—can become a bottleneck for write-heavy workloads. At the data structure layer, the "performance tax" is not uniform; cache-friendly structures (e.g., B+ trees) absorb the overhead of memory safety checks better than pointer-chasing structures (e.g., linked lists). For full systems, there is an "engineering-performance" trade-off: MTE offers high portability and modest overheads (<10%) but provides only probabilistic safety, whereas CHERI offers deterministic, fine-grained protection at the cost of higher porting complexity and performance penalties (20%–60%).

To summarize, our study serves as a bridge between architectural research and industrial database engineering. By quantifying the performance-safety trade-offs of modern hardware, we provide a concrete roadmap for building the next generation of "reliable/secure-by-default" data infrastructure. Our work demonstrates that hardware-assisted safety is finally approaching the maturity required for production database systems, offering a viable path to resolving the long-standing database trilemma.

Overall, this paper makes the following contributions:

- **Empirical bug study:** We quantify the prevalence of memory-safety-related Common Vulnerabilities and Exposures (CVEs) and bugs in open-source database systems, showing they remain a substantial share of reported issues.
- **Hardware-assisted memory safety extensions:** We port six data structures and six database systems. We detail the best practices and pitfalls of using the MTE and CHERI extensions.
- **Cross-layer experimental analysis:** We evaluate the performance across four layers: *(i) microarchitecture*, *(ii) data structures*, *(iii) database systems*, and *(iv) compilers, runtime, and OS*.
- **The database trilemma synthesis:** We formalize and evaluate the trade-offs between safety, performance, and portability.
- **Open-source evaluation framework:** To facilitate further research into secure and reliable data infrastructure, we make our entire evaluation pipeline, including the "MTE-tagged" and "CHERI-fied" database systems source code, publicly available.

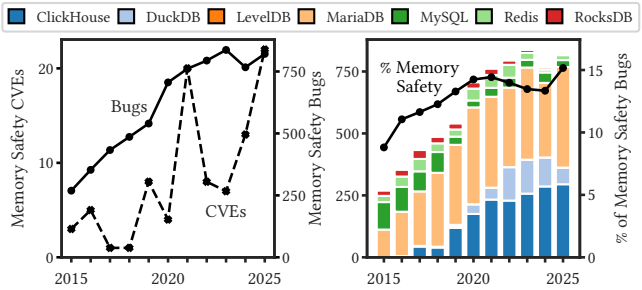
2 Background and Motivation

We first establish why memory safety violations are a pressing concern for database systems, and why existing approaches fail to

	Total issues	Memory safety (%)	Implication							
			Cr	DC	IQ	RE	DoS	Co	Rp	DL
DuckDB	5,981	549 (9.2%)	493	0	13	4	2	1	0	1
Redis	3,433	393 (11.4%)	370	2	0	9	1	0	0	0
RocksDB	2,432	227 (9.3%)	205	3	0	6	0	0	0	0
ClickHouse	22,295	1,681 (7.5%)	1,434	6	57	24	3	23	0	0
MySQL	21,503	614 (2.9%)	476	6	10	42	1	1	1	0
LevelDB	306	13 (4.2%)	12	0	0	1	0	0	0	0
MariaDB	30,536	3,290 (10.8%)	2,905	41	60	70	1	1	1	0
CVEs	1,448	92 (6.4%)	60	0	1	0	3	0	0	0
Total	**87,934**	**6,859 (7.8%)**	**5,955**	**58**	**141**	**156**	**11**	**26**	**2**	**1**

Cr:Crash; **DC**:Data corruption; **IQ**:Incorrect query results; **RE**:Resource exhaustion
DoS:Denial of service; **Co**:Concurrency hazard; **Rp**:Replication issue; **DL**:Data loss

Table 1: Memory safety bug and CVE analysis (2015–2025).



(a) Number of new bugs/CVEs per year (b) Breakdown per database system

Figure 3: Time-based memory safety bug and CVE analysis.

resolve the tension between safety, performance, and portability.

2.1 Memory Safety in Database Systems

Despite the performance benefits of manual memory management, database systems written in unsafe languages (e.g., C/C++) are inherently exposed to spatial memory safety violations (e.g., buffer overflows, out-of-bounds accesses) and temporal violations (e.g., use-after-free, double-free) [93, 126, 134], whose consequences go well beyond simple crashes. A single erroneous memory access can cause *silent data corruption* by overwriting adjacent records before they are flushed to storage, produce *incorrect query results* by corrupting in-memory index structures, *exhaust system resources*, introduce *concurrency hazards* by corrupting shared metadata, *expose sensitive user records* to unauthorized parties, or allow unauthorized transactions—ultimately threatening the correctness and durability guarantees that database systems are designed to provide.

To quantify the prevalence of memory safety bugs in real-world database systems, we conduct a systematic study of publicly reported CVEs and bug reports in the last decade (2015–2025) across seven open-source database systems—ClickHouse, DuckDB, LevelDB, MariaDB, MySQL, Redis, and RocksDB—spanning different deployment models and storage engines. The volume of bugs and CVEs (Table 1), and the non-expert nature of bug report writers render manual classification impractical and simple keyword matching unreliable.

Instead, we propose using a Large Language Model (LLM) to classify bug reports and CVEs. We employ a locally hosted GPT-OSS 120B model on an NVIDIA H100 GPU to classify each report into *bug*, *programmer error*, or *other*, as some bug trackers include additional reports (e.g., user errors, feature requests, and more). Given the higher selectivity, we consider all CVEs to be bugs. Further, the LLM determines whether a bug is memory-safety-related and, if so, identifies its primary operational implication, given a set of options. Each classification is repeated three times independently. We treat a bug as a confirmed memory safety issue only when *all* three repetitions agree. Further details on the methodology are provided in Appendix B.

Limitations. We do not aim to establish a new state-of-the-art vulnerability classifier, but to provide a system-specific overview complementing related work [20, 81, 127, 130] using more robust pipelines and frontier models. Our pipeline has several limitations, and we report its results as *motivating estimates*, not exact counts. First, we run a quantized model without ground-truth validation. Second, requiring agreement across all three runs excludes inconsistently flagged borderline bugs. However, as the model tends to repeat errors [28], agreement provides less confidence than three independent judgments. Third, we classify only issue-tracker reports and CVEs, excluding fixes landed without corresponding issues; our claims therefore concern memory-safety *reports*, not all memory-safety bugs. Finally, MySQL redacts security-sensitive tracker issues, which may affect its memory-safety share. The per-year timeline is also affected by project age, the model's familiarity with recent terminology, and gaps in CVE enrichment; we therefore interpret it as evidence of sustained presence, not a precise trend.

Our analysis yields three key findings: (i) Memory safety bugs affect all database systems. Even recent systems (e.g., DuckDB, ClickHouse) are not immune, highlighting that memory-safety violations remain prevalent despite good practices and modern software engineering techniques. (ii) Memory-safety issues remain a substantial source of bugs in these systems today: through the period, they consistently account for a non-trivial share of all reported bugs and CVEs every year (Figure 3), despite significant advances in testing, fuzzing, and bug-finding tools [36, 48, 52, 66, 68, 117–119, 123, 152, 153]. By contrast, large vendors such as Microsoft and Android identified memory safety as the source of ≈ 70% of security issues [8, 9, 112] and report a *declining* share after deploying mitigations, e.g., Android from 76% to 24% between 2019–2024 [135]. We observe no comparable decline in our dataset. (iii) Memory safety violations can cause diverse harm, as shown in Table 1. The most common implication is a system crash (≈87%), followed by resource exhaustion, and incorrect query results.

2.2 Scope: The Database System Trilemma

Memory safety solutions are governed by three competing dimensions: *performance overhead*, *protection coverage*, and *portability*. High-performance systems typically reject solutions that introduce non-trivial latency. Consequently, developers often employ best-effort safety models that prioritize execution speed over exhaustive protection. However, the primary determinant for adoption is often the engineering effort required for integration. For large-scale systems, the cost of manual refactoring often outweighs the benefits of improved memory safety. Several software-based approaches have been proposed to prevent memory safety violations [17, 21, 27, 39–41, 54, 55, 60, 62, 67, 73–75, 85, 86, 102, 104, 105, 120–122, 124, 128, 139, 146, 150], with all facing the challenge of reconciling the tensions within this trilemma. Their runtime overheads remain the most significant hindrance to production deployment. The alternative of memory-safe languages shows promising results

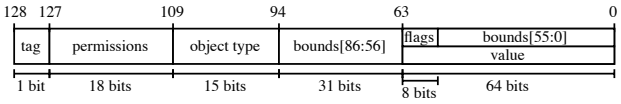


Figure 4: CHERI Morello capability format.

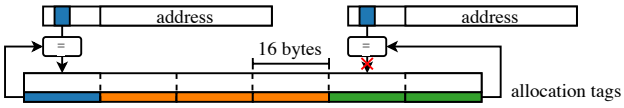


Figure 5: MTE tag checking mechanism.

both in new projects [35] and rewrites of core production components [65]. However, rewriting mature systems with millions of lines of highly optimized code remains a substantial and often prohibitively costly undertaking. In contrast, hardware-assisted memory safety promises significantly reduced overhead and, by extension, broader industry adoption. In this paper, we conduct a detailed empirical analysis of two such extensions—ARM MTE and CHERI—designed to provide hardware-backed memory-safety.

3 Architectural Analysis

After establishing hardware-assisted memory safety as a promising direction, we describe and evaluate the microarchitectural foundations of two modern memory safety extensions implemented in research and production hardware (CHERI (§3.1) and MTE (§3.2)) to realize the costs and constraints that any high-level system will inherit. These two extensions differ in maturity: MTE ships in production silicon today, whereas CHERI is available only on research hardware. We also analyze Intel MPK in Appendix A.1.

Experimental setup. We conduct our experiments on the Morello Evaluation Platform [57], a prototype implementing CHERI, and an Ampere 1a server, a commercially available server. We run all experiments with synchronous MTE (§3.2), as this is the only mode the Ampere 1a supports [69]. For CHERI, we use Morello LLVM [15]; when compiling in *purecap* mode (§3.1), we link against a capability-aware port of the Musl C standard library [16]. For MTE, we use upstream LLVM and glibc with MTE enabled. The detailed testbed is described in our supplementary material (Appendix A). We evaluate instructions by executing them 1.6×10^9 times in an unrolled loop of 16 instructions. We measure *(i) Latency* with data dependencies between instructions and *(i) Throughput* without dependencies.

3.1 CHERI

Mechanism. CHERI [142] replaces 64-bit pointers with 128+1-bit hardware-enforced capabilities that encode a memory address, the accessible boundaries, its permissions and object type, and a validity bit (Figure 4). Capabilities are *unforgeable* and *monotonic*: overwriting a capability with arbitrary data clears its validity bit. New capabilities can only be derived by narrowing existing ones. Morello implements CHERI for Armv8.2-A, storing the validity bit alongside data in memory. CHERI protects against spatial violations and enables control-flow integrity, while temporal safety can be achieved through capability revocation [45, 140, 143]. Code can be compiled in two modes: *purecap*, where every pointer is a capability, or *hybrid*, where only explicitly annotated pointers are.

Microarchitectural Analysis. We evaluate a representative set of Morello instructions in Table 2. For memory operations, we distinguish three groups: plain loads/stores (ldr, str) moving 64-bit integers via capability-based addresses, *cap* variants loading or storing 128+1-bit capabilities, and *in place* variants that repeatedly access the same address rather than advancing the pointer.

Category	Instruction	Lat	Tp
Memory	ldr	4.00	2.00
	str, str (in place)	—	2.00
	ldp, stp,str (cap), ldp (cap)	—	1.00
	ldr (cap)	4.46	2.00
	stp (cap)	—	0.50
	ldxr, ldxr (cap)	—	0.18
	str (cap, in place), stp (in place)	—	0.33
	stp (cap, in place)	—	0.17
	stxr	—	0.13
	stxr (cap)	—	0.11
	cas (cap), swp (cap)	—	0.07
Conversion	cvtd_tocap, cvtp_tocap, cvt_tocap	2.00	1.00
	cvtd_toptr, cvtp_toptr, cvt_toptr	1.00	1.00
Cap Ops	add (cap), sub (cap), orrflgs, bicflgs, gcperm, gctype, gcvalue, gcflgs, gcseal, gctag, clrperm, cpy	—	2.82
	cfhi, cthi	1.00	2.82
	cvtdz_tocap, cvtpz_tocap, cvtz_tocap, cpytype, cpyvalue, rrlen, rrmask, alignd, alignu	1.00	1.00
	gcbase, gclen, gclim, gcoff, scoff, scflgs, scvalue, scbnds, scbndse, seal, cseal, unseal	—	1.00
Cap Checks	chkss, chkssu	—	0.50
	chktgd, chkeq	—	1.00

Lat: Latency (cycles), higher is better; **Tp**: Throughput (inst/cycle), lower is better

Table 2: Morello CHERI instructions

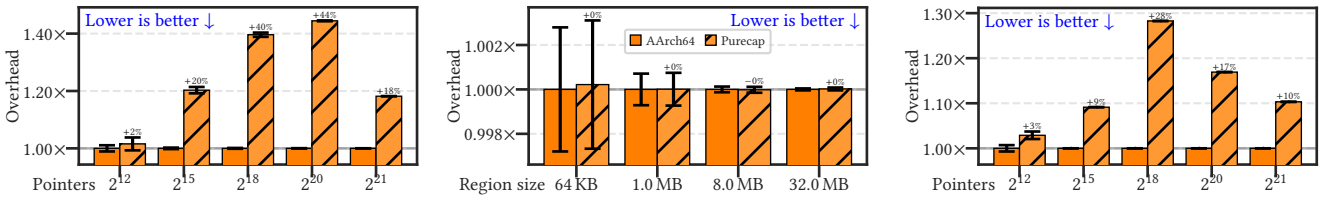
Notably, loading one capability achieves the same throughput as a 64-bit integer load (2 inst/cycle), even though the external data bus remains 64-bit wide. Loading two capabilities and two integers both reach ≈1 op/cycle. Stores, however, scale with the amount of data written; *stp (cap)* reaches only 0.5 inst/cycle, half that of *stp*.

Memory. We benchmark Morello in two configurations: *(i) AArch64*, running unmodified code and *(ii) Purecap*, where all accesses use fine-grained capabilities. We perform a set of benchmarks for different access patterns: Figure 6a presents pointer chasing in a randomized list, Figure 6b reflects the memory-region copy, and Figure 6c shows the atomic accesses, where three readers and one writer operate on pointers/capabilities concurrently. In Figure 6a, the Purecap list contains half the entries of the AArch64 variant, as capabilities occupy 128 bits. In Figure 6, we observe a significant overhead, caused by the change from pointers to capabilities, which decreases at larger working sets: once accesses miss in cache, a full 64-byte cache line accommodates both a pointer and a capability, equalizing the cache-miss penalty. For copies, we see no notable overhead, as both variants copy the same amount of data, and the baseline performs checks using the Default Data Capability (DDC) [89, §1.3.5]. In Figure 7a, concurrent tag writes cause a significant throughput drop; writing the validity bit invalidates the entire cache line, since tags are stored inline with data, forcing readers to re-fetch both. As a result, all configurations converge to the RAM access throughput.

Takeaway: CHERI doubles the pointer size, thus halving the cache capacity for pointer-heavy workloads. While the hardware manages single-capability loads efficiently, concurrent metadata updates incur overhead from cache line invalidations.

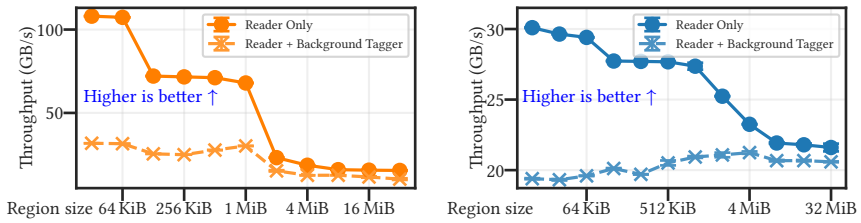
3.2 MTE

Mechanism. Arm's Memory Tagging Extension (MTE) [88][90, D.10] assigns tags to memory regions and pointers and checks if they match on every access. Memory is divided into aligned 16 byte



(a) Pointer chasing. (b) Memory copy. (c) Atomic operations.

Figure 6: Linear and non-linear memory access patterns with and without CHERI.



(a) CHERI validity tag. (b) MTE allocation tag.

Figure 7: Load throughput with/without a background thread writing memory tags.

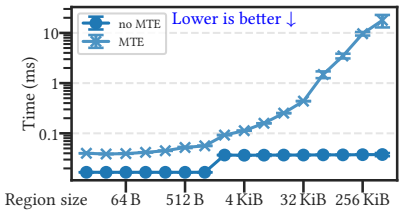
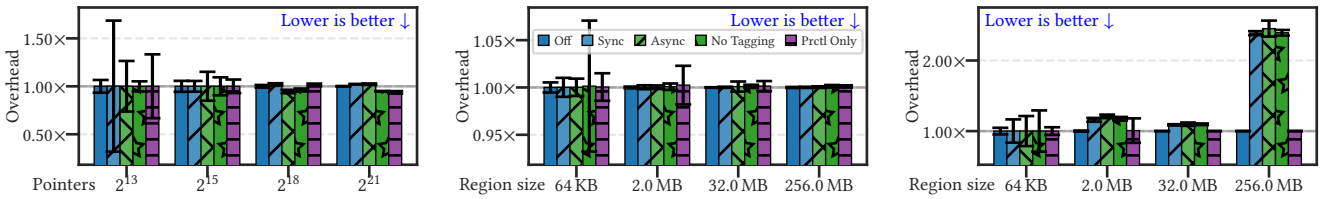


Figure 8: Glibc malloc time with and without MTE tagging.



(a) Pointer chasing. (b) Memory read. (c) Memory write.

Figure 9: Linear and non-linear memory access patterns with and without MTE. Variance is high for the smallest working sets because they fit entirely in cache, resulting in short execution times dominated by fixed overheads and noise.

Inst	Description	Lat	Tp
irg	Generate random tag.	3.41	0.29
addg	Add with tag, skipping excluded tags.	0.98	2.04
subg	Subtract with tag, skipping excluded tags.	0.98	2.04
subp	Subtract pointer, preserving the tag.	0.98	3.88
subps	Subtract pointer, preserving the tag, setting flags.	0.98	1.94
stg	Store allocation tag.	—	1.02
st2g	Store two allocation tags.	—	0.51
stzg	Store allocation tag, zeroing.	—	1.02
stz2g	Store two allocation tags, zeroing.	—	0.51
stgp	Store allocation tag and pair of registers.	—	1.02
ldg	Load allocation tag.	—	1.99

Lat: Latency (cycles), higher is better; **Tp**: Throughput (inst/cycle), lower is better

Table 3: MTE instructions

Mode	Off	Sync	Async	No Tagging	Prctl Only
read	9.5 M	10.2 M	9.5 M	10.5 M	14.3 M
write	179.4 M	**468.1 M**	**467.4 M**	**468.2 M**	184.0 M

Table 4: Data Cache Misses for MTE.

tag granules, each with a 4-bit *allocation tag* stored separately or alongside the data [69]. Pointers embed a *logical tag* in bits 56–59. On memory access (Figure 5), the hardware checks that tags match or triggers a fault: synchronously at the offending instruction, or asynchronously via a status register checked on kernel entry. This design has two consequences: By assigning different tags to adjacent or freed regions, MTE protects against spatial and temporal violations, but not intra-object overflows. Additionally, MTE limits the number of tags and only provides probabilistic safety.

Microarchitectural Analysis. Table 3 lists the performance of MTE instructions. In Figure 9, we measure MTE overhead across

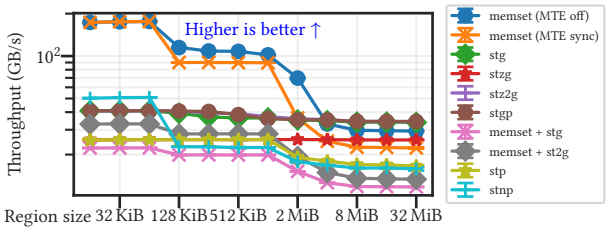


Figure 10: Memory and tag initialization.

five configurations: *Off*, *Sync* (faults on offending instruction), *Async* (faults on kernel entry), *No Tagging* (sync, zero tag), and *Prctl Only* (sync, MTE disabled for accessed pages). On the Ampere 1a, allocation tags are stored alongside data; reads therefore fetch both tag and data in a single operation, incurring no overhead (Figures 9a and 9b). Stores (Figure 9c) must load the existing allocation tag before writing; this nearly triples the number of D-cache misses under MTE (179.4 M → 468.1 M, Table 4). As our server only implements synchronous MTE, *sync*, *async*, and *no tagging* perform similarly. Figure 7b shows memory throughput with a second thread changing allocation tags, showing a similar trend as CHERI.

Takeaway: MTE provides probabilistic safety with an asymmetrical performance profile: reads remain efficient, but writes incur a higher performance penalty.

Memory Allocations. We evaluate glibc's ptmalloc implementation, with and without MTE, by allocating/freeing chunks of various sizes in Figure 8. Without MTE, allocation complexity is $O(1)$, with

Metric	memset	stzg/stgp	stnp	stp
Cycles	2.48×10^9	1.6×10^9	3.47×10^9	3.22×10^9
Cache Miss Rate	95.87%	**0.08%**	24.79%	24.80%

Table 5: Memory initialization perf stats (32MB).

a slow-down above 1 KiB where glibc switches from extending its contiguous heap via *sbrk* to obtaining separately mapped memory via *mmap*. With MTE, small allocations lose the fast *sbrk* allocation path, and the allocation time grows linearly ($O(n)$) as the entire region must be tagged, evicting cache lines in the process. Thus, large, short-lived allocations may benefit from different protection mechanisms (e.g., guard pages).

When initializing memory, choosing the right primitive also matters for performance. Figure 10 compares several initialization strategies. For buffers that fit in data caches, *memset* performs the best. For larger allocations, tag-setting instructions (*stzg*, *stgp*) outperform plain stores (*stp*, *stnp*, *memset*) because the CPU implicitly fetches the allocation tag before stores, leading to 95.87 % D-cache miss rate (Table 5). Tag-setting instructions write the tag without the implicit fetch, reducing the miss rate to 0.08 %.

4 Analysis of Data Structures

Building upon our analysis, we now investigate how MTE and CHERI overheads propagate to the core data structures of modern database systems, chosen within a representative taxonomy.

4.1 Data Structures Taxonomy

To this end, we propose a use-case-driven taxonomy to ensure coverage of the distinct memory access patterns found in practice.

Primary access to data. Primary index data structures express relationships across large parts of the data and are used to improve the performance of typical operations. For Key-Value Store workloads, systems typically use *Log-Structured Merge (LSM)* trees or *hash tables*. *B*-trees* and *tries* are common for transaction-oriented systems, whereas analytics-oriented systems usually opt for *block range* indexes. Finally, graphs are typically represented in *adjacency lists* using the Compressed Sparse Row (CSR) format.

Temporary buffering. Database systems use additional structures to hold temporary data. Temporary data can be buffered before writing to storage, especially in write-heavy workloads, or when executing long-running, complex queries. Key-Value Store systems commonly use *Skip lists* to buffer writes in memory, while transactional systems usually perform writes directly in the primary data structure. Analytical workloads require complex selections and joins, which are typically performed using *hash tables*.

Internal organization. Database systems employ auxiliary data structures internally to organize transient data and manage state and operations (e.g., *queues*). For our analysis, we choose established representatives of the above categories, listed in Table 6, along with their operations and memory access types. For primary indexes, we select the B+ tree from the Leanstore [78] implementation that uses optimistic lock coupling, and the Adaptive Radix Tree (ART) [79]. For temporary data buffering, we consider the cache-line-aligned hash table (CLHT) [34] and LevelDB's Skip list [64]. Finally, we examine Boost's implementations of a queue and a linked list.

4.2 Porting to Memory Safe Hardware

CHERI. For the CHERI `purecap` target, the compiler handles most of the portable C/C++ code. The compiler natively inserts capability-aware atomic operations, used in lock-free data structures.

Three classes of issues require manual intervention. First, CHERI enforces 16-byte capability alignment, disrupting cache-aware designs designed to prevent false sharing or pack pointers densely. Second, traditional bit-wise pointer manipulation for custom alignment discards capability metadata, triggering runtime violations. Third, pre-allocating a large region and subdividing it at runtime holds only a capability covering the entire range by default, providing no sub-region safety. We resolve these issues by replacing alignment-dependent code (e.g., in Boost's queue) with CHERI-native alignment functions, and narrowing the returned capability to each allocated block at the point of subdivision.

MTE. Porting to MTE requires no changes beyond the memory allocator. For data structures that rely on the system allocator, we use an MTE-enabled allocator (e.g., glibc's Ptmalloc); the data structure code itself remains intact. Turning on tagging mandates glibc to support memory tagging (>2.33) and to set the *GLIBC_TUNABLES* environment variable to *glibc.mem.tagging=3*.

However, high-performance data structures often define their own allocators. We port these by modifying alloc/free functions to align pointers to 16-byte granule boundaries, assign a random tag to the pointer (using *irg*), and, for larger allocations, iterate over the allocated memory range and set the allocation tag (with *stg*).

Another non-trivial case arises in data structures that encode links between elements as custom indices rather than pointers—our B-tree, for instance, stores parent-to-child links as Page IDs (PIDs) and reconstructs the pointer on every traversal. To minimize changes to the data structure design, we first reconstruct the pointer and then fetch the allocation tag using the *ldg* instruction.

Takeaway: Retrofitting data structures for hardware safety extensions requires careful porting due to the pointer length and alignment assumptions. Porting to MTE is simpler, as most modifications are in the allocators, whereas CHERI requires invasive refactoring to maintain capability provenance and alignment.

4.3 Experimental Analysis

Methodology. We evaluate the performance overhead of MTE and CHERI on each data structure operation using microbenchmarks. Unless noted otherwise, all experiments use the same experimental setup as described in §3. We plot the latency relative to the respective native baseline in Figure 11. We report configurations whose baseline and safety-enabled 95% confidence intervals overlap as not statistically significant and mark them with an asterisk. Table 6 lists each structure's supported operations (insert, read, update, scan) and their memory access patterns. For the queue, we pair a producer and consumer thread and measure the latency of a single enqueue/dequeue round-trip. We evaluate concurrent data structures (B+Tree, CLHT, and queue) across thread counts, *i.e.,* 1, 8, and 64 on the MTE platform and 1, 2, and 4 on the 4-core Morello board.

B+Tree. The B+Tree shows a moderate overhead (Figure 11a), ranging up to 25% on MTE and from 13% to 33% on CHERI when single-threaded. Inserts and updates are the most affected operations, as pointer chasing combined with optimistic lock coupling

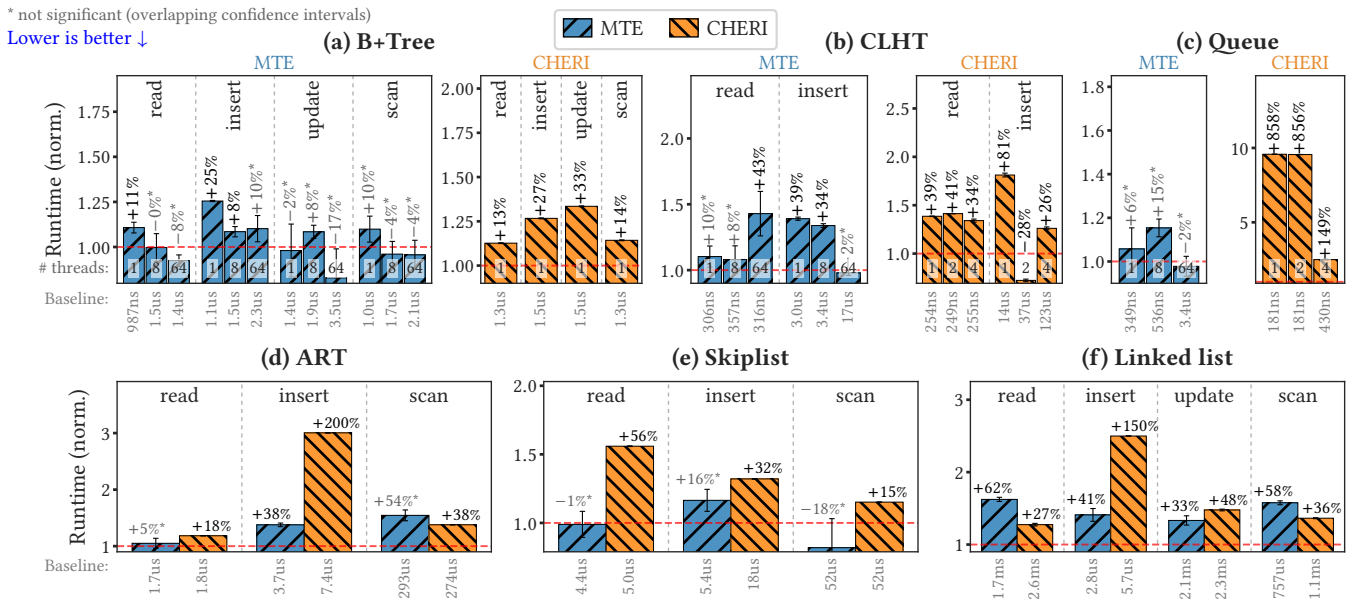


Figure 11: Overhead of MTE and CHERI on commonly-used data structures in database systems.

	Structure	Op	Description
Primary	B+Tree	read	Follow fences down the tree *(Rnd)*
		scan	Read *(Rnd)* and iterate in last level node(s) *(Seq)*
		insert	Find location *(Rnd)* then emplace *(CAS W)* and potential rebalancing *(Rnd)*
		update	Find location *(Rnd)* then replace *(CAS W)*
	ART	read	Follow partial keys down the tree *(Rnd)*
		scan	Read *(Rnd)* and iterate in last level node(s) *(Seq)*
		insert	Find location *(Rnd)* then emplace *(W)* and potential node growth/rebalancing *(Rnd)*
Temporary	CLHT	insert	Compute hash then emplace *(Rnd + CAS W)*
		read	Compute hash then retrieve *(Rnd)*
	Skiplist	read	Use index and access element *(Rnd)*
		scan	Use index *(Rnd)* then iterate *(Seq)*
		insert	Use index *(Rnd)* then emplace *(W)*
Management	Queue	produce	Put element and advance head (Rnd CAS)
		consume	Get element and advance tail (Rnd CAS)
	Linked list	read	Iterate through the list *(Rnd)*
		scan	Iterate through for each element *(Rnd)*
		insert	Emplace at the tail *(W)*
		update	Find location *(Rnd)* then replace *(W)*

Table 6: Data structures operations by memory access types *Random (Rnd), Sequential (Seq), Write (W), and Atomic (CAS)*

requires expensive atomic synchronization. Scans, however, are efficient (10% MTE, 14% CHERI), since the large 4 KiB nodes frequently contain the entire 100-key scan range, minimizing inter-node traversals. With rising thread counts, the MTE overhead shrinks and, at 64 threads, is no longer statistically significant for any operation: cache-coherence traffic and lock-coupling contention occupy a larger portion of the runtime and hide the tagging cost. We do no report multi-threaded CHERI configuration: the B+Tree uses C++ exceptions to implement optimistic lock coupling, and we could not get exception unwinding working under purecap (§6).

ART. For ART, MTE overhead is 5–38%, while CHERI reaches 18–200%, with the gap widening for write-heavy operations (Figure 11d). Insertions are the most expensive (38% MTE, 200% CHERI) as node growth and rebalancing trigger extra random accesses and amplify write penalties. Scans incur moderate overhead on CHERI (38%) when traversal crosses node boundaries to locate siblings, whereas intra-node sequential access remains largely unaffected.

CLHT. In CLHT operations (Figure 11b), MTE overhead is modest at up to 43%, while CHERI is more pronounced at 26–81%, reflecting the sensitivity of lock-free designs to atomic operation costs. Read-only workloads show minimal MTE impact at 1–8 threads and 43% at 64 threads, but incur 34–41% overhead under CHERI, where even pointer loads carry a validation cost. Inserts are the most expensive (up to 39% MTE, 81% CHERI), as they rely on multiple atomic compare-and-swap operations, more costly under CHERI.

Skip list. The skip list shows no statistically significant MTE overhead, while CHERI adds 15–56% (Figure 11e). Although the multi-level index reduces search depth, random pointer chasing across levels still incurs a 56% penalty on CHERI. Insertions cost 32% on CHERI, as maintaining the skip list multiplies pointer updates.

Queue. The queue (Figure 11c) starkly contrasts the two extensions. MTE almost matches the baseline, as the queue stores integer values that bypass tag checking. Conversely, CHERI incurs bigger overhead (up to 858% in the worst-case scenario), since pure-capability mode protects every internal pointer and the lock-free produce/consume logic requires frequent atomic writes, an operation heavily penalized by CHERI. However, in a more realistic scenario with more threads, contention on the head and tail represents a bigger portion of the latency, and the CHERI overhead drops to 149%.

Linked list. For the linked list (Figure 11f), MTE overhead (33–62%) is primarily due to random pointer chasing during full-list traversals, where each node access may incur a cache miss. Under CHERI, insertions peak at 150% due to heavy write penalties, while read and scan overheads (27–36%) are comparable to MTE, as the bottleneck shifts from capability validation to memory latency. The non-linear scaling we observe for allocation-heavy operations stems from the $O(n)$ allocation cost of MTE tagging (§3).

Type	System	Workload	C/C++ LoC	LoC changed MTE	LoC changed CHERI
In-memory KVS	Redis [114]	YCSB [32]	281 K	+61/-22	+140/-33
Persistent KVS	LevelDB [64]	YCSB [32]	93 K	+13/-12	+12/-9
OLTP (server)	MySQL [108]	TPC-C [4]	4.07 M	+56/-5	—
OLTP (embedded)	SQLite [37]	TPC-C [4]	404 K	+15/-3	+35/-3
OLAP	DuckDB [113]	TPC-H [5]	986 K	+225/-10	—
Graphs	LadyBugDB [44]	LDBC SNB [77]	370 K	+63/-2	+55/-10

Table 7: Database systems ported to MTE and CHERI.

Takeaway: The overhead of MTE and CHERI is dictated by the data structure memory access patterns. Cache-aligned, contiguous data structures, *e.g.*, B+ Trees, incur less overhead. Conversely, lock-free structures and those that rely heavily on pointer chasing experience more performance degradation.

5 Database Systems

After seeing the overhead at the data structure level, we analyze the impact of CHERI and MTE across a representative set of database systems to examine whether these costs remain acceptable when MTE and CHERI are applied to full-fledged database systems under realistic workloads and how much engineering effort porting each system requires. Our ports are best-effort: we are not the maintainers of these systems, so our changes may still contain bugs and could be sub-optimal. Table 7 lists the chosen database systems, the workloads used, and the modified Lines of Code (LoC) required to port them to MTE and CHERI. All experiments in this section use the experimental setup as described in § 3. Note that all systems work natively out of the box on Arm architectures (*AArch64*).

Experimental parameters. We sweep both the working-set size and the thread count. Working sets span three points an order of magnitude apart: 100 K–10 M records for LevelDB, 10 K–1 M keys for Redis, 10 K–10 M rows for MySQL, 1–50 warehouses for SQLite, and scale factors 1–100 for DuckDB and LadyBugDB. Thread counts are bounded by the available cores: 1, 8, and 64 on the 192-core Ampere 1a, and 1, 2, and 4 on the 4-core Morello board.

5.1 In-Memory Key-Value Store (Redis)

Redis is a popular in-memory key-value store that caches clients' data in an ordered fashion. Redis uses a hash table with separate chaining for collision handling and relies on *jemalloc* for superior performance and memory utilization in database workloads [43].

Porting. To port Redis to CHERI, we use the purecap port of Musl libc, with static linking to avoid mixed-ABI symbols. The main code changes involve replacing pointer casts to 64-bit integer types (e.g., *unsigned long*) with *(u)intptr_t* to preserve capability metadata, and ensuring 16-byte capability alignment. One non-trivial issue arises when Redis exploits the contiguity of *argv[0]* through *argv[argc]* to write to a shared buffer via *argv[0]*. This trick is forbidden by the capabilities model, as each pointer is bound to its own allocation. We fix this by restricting the access to *argv[0]* alone.

For MTE, we port the `jemalloc` allocator used by Redis. When allocating, `jemalloc` generates a new tag using *irg* and assigns it to the region, ensuring adjacent regions receive different tags. It reserves tag 0 for freed regions, and tags 1–15 for used memory.

Results. We use the YCSB workloads [32] with 100 K operations, sweeping the working set from 10 K to 1 M keys; since the single-threaded *redis-server* saturates fast, we run it with one client. The overheads are insensitive to the working-set size, so Table 8a reports the average across the sweep: both extensions introduce modest average overheads (1% for MTE, 5% for CHERI). This can be attributed to Redis's relatively low operational throughput, which limits the impact of per-operation safety checks on end-to-end latency.

Higher is better ↑			YCSB-A		YCSB-B		YCSB-C		YCSB-D		YCSB-E		YCSB-F	
Store	Size	thr	base	prot.	base	prot.	base	prot.	base	prot.	base	prot.	base	prot.
(a) Redis														
MTE +1%	all	1	16K	16K	15K	15K	15K	15K	14K	14K	357	351	11K	11K
CHERI +5%	all	1	24K	22K	25K	25K	26K	24K	25K	23K	744	680	18K	17K
(b) LevelDB														
MTE	100K	64	152K	144K	624K	623K	723K	724K	685K	688K	371K	284K	156K	147K
	1M	1	26K	23K	54K	49K	88K	77K	60K	55K	9K	6K	31K	27K
mem **+7%**	1M	8	149K	146K	426K	391K	812K	770K	474K	442K	55K	40K	145K	141K
	1M	64	136K	132K	562K	558K	853K	907K	609K	611K	198K	184K	135K	130K
	10M	64	67K	73K	314K	300K	411K	378K	465K	542K	163K	139K	40K	52K
disk **+1%**	1M	64	115K	116K	549K	525K	861K	895K	580K	585K	183K	176K	123K	117K
CHERI	100K	4	43K	29K	44K	32K	39K	27K	48K	35K	1K	820	31K	20K
mem **+53%**	1M	1	32K	23K	82K	54K	102K	72K	83K	60K	4K	3K	30K	21K
	1M	2	38K	29K	87K	48K	97K	41K	83K	54K	3K	1K	32K	23K
	1M	4	37K	24K	42K	31K	40K	27K	44K	33K	1K	815	27K	17K
disk **+48%**	1M	4	36K	25K	41K	30K	41K	28K	43K	31K	1K	803	27K	18K

Overhead: 0.8× 1× 1.5× 2.2×

Table 8: Overhead of MTE and CHERI on Redis and LevelDB.

5.2 Persistent Key-Value Store (LevelDB)

LevelDB is a persistent key-value store library that uses a Log-Structured Merge Tree (LSM). Writes are buffered in skiplist-based MemTables, and eventually flushed to storage in SSTables.

Porting. For CHERI, we use the same Musl-based purecap toolchain as Redis, with LLVM's *libc++* library provided by Morello. We disable automatic linking to the *stdlibc++* and manually add the required libraries. For MTE, LevelDB only needs a MTE-aware *glibc* for the memory allocator and setting *glibc.mem.tagging=3*. Overall, LevelDB requires no code changes beyond toolchain configuration.

Results. We execute the YCSB workloads with 1 M operations across the full working-set (100 K–10 M records) and thread-count sweeps. Across workloads (Table 8b), MTE incurs a mean overhead of 7%, largely stable across working-set sizes and thread counts, while CHERI averages 53%, with the largest penalties on the insert-heavy workloads (D, E), confirming our skiplist microbenchmarks (§ 4). MTE results at 10M keys, 64 threads that report a speedup are within the confidence interval of the baseline. We compare the *in-memory* results to a *disk* configuration, where the SSTables reside on an SSD and the block cache is capped at 1% of the dataset size, forcing cache replacement (disk rows in Table 8b). The out-of-core overhead is a bit inferior to the in-memory results: MTE's overhead decreases from ≈7% to 1% , while CHERI incurs 48% overhead on disk versus 53% in memory, indicating that I/O operations have modest impact on the overhead of MTE/CHERI.

5.3 Transactional Database Server (MySQL)

MySQL is a transactional (OLTP) relational database server that enables applications to store structured data and execute complex SQL queries. It uses a B+ Tree for indexingand a hash-table-based buffer manager to map Page IDs to memory locations.

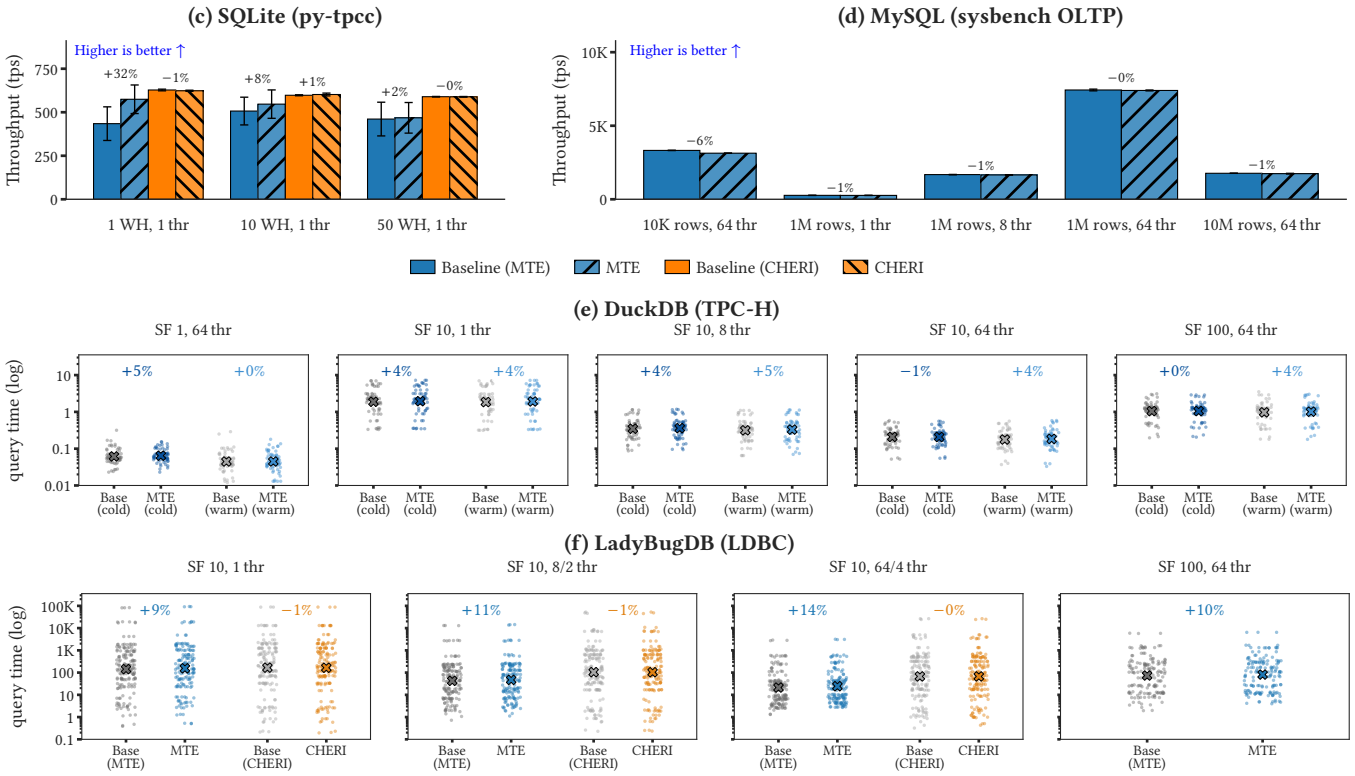


Figure 12: Overhead of MTE and CHERI on database systems (MySQL and DuckDB could only be ported to MTE).

Porting. MySQL relies on the OpenSSL library to provide cryptographic functionalities to connections. At the time of writing, no CHERI purecap port of OpenSSL exists, preventing us from porting MySQL to CHERI. For MTE, we patch MySQL's memory allocator to tag allocated regions before returning tagged pointers to the callee and re-tag them on free, invalidating dangling pointers and hardening against use-after-free. To detect double-free errors, we probe the region before re-tagging, which triggers a fault when a free is repeated. As with any MTE-based allocator, tag collisions remain possible due to the limited tag space; quarantining freed memory can mitigate this at the cost of higher memory usage.
Results. We execute a TPC-C-like workload [4] sweeping the table size from 10 K to 10 M rows and the client count from 1 to 64, using `sysbench_oltp` [71] load generator, and measure throughput after 60 seconds. As shown in Figure 12d, MTE reduces MySQL's throughput by at most 6%—on the smallest, fully cached table under 64 clients—and by ≤1% in every other configuration.

5.4 Embedded Database System (SQLite)

SQLite is an embedded relational (OLTP) database system. It provides a SQL engine that interacts directly with disk files.and uses a B+Tree for table storage, with a default block size of 4 KiB.
Porting. For CHERI, we follow the same process as in Redis. A notable challenge lies in SQLite's allocator, which adds 64 bits of metadatato each allocation and returns a pointer to the region.Under CHERI, this pointer retains capabilities over the entire region , with the metadata, allowing buggy code to corrupt the allocator's state. Restricting the capability to the user region prevents this, but requires the allocator to maintain a separate capability for metadata access. To minimize code changes, we retain the unrestricted pointer returned by the allocator, which allows metadata access.

For MTE, since SQLite relies on the system allocator, we use MTE-enabled glibc and avoid code modifications. However, MTE initialization via *prctl* must occur exactly once per process. Standalone database systems can perform this initialization at startup; for embedded libraries such as SQLite, initialization must be handled either by the host application or by the library itself, in which case it should first check the active MTE state to avoid conflicts.
Results. We execute a synthetic transaction workload on SQLite using tpcc-python [111], with 1–50 warehouses and measure transaction throughput after 60 seconds. The throughput differences of both the MTE and CHERI ports remain within the measurements' confidence intervals at all sizes, as depicted in Figure 12c.

5.5 Analytical Database System (DuckDB)

DuckDB is an embedded analytical (OLAP) system optimized for complex analytical queries. It uses a columnar format, with fixed-size row groups and min-max indexes to skip unnecessary blocks.
Porting. DuckDB uses external dependencies bundled via CMake. We define a toolchain file that realizes most of the build system modifications. DuckDB's reliance on functions absent from Musl libc (e.g., *strtoll_l()*) requires a compatibility shim layer that translates missing functions to their Musl counterparts. Another three classes of issues require further manual fixes. First, five locations in the codebase assume pointers are exactly 8 bytes; we update them to accommodate CHERI's 16-byte pointers, while accounting for silent assumptions in the buffer manager where *size_t* and pointer sizes are used interchangeably. Second, integer pointer arithmetic

Feature	CHERI	MTE
Compiler Optimizations		
Ptr Aliasing	Disabled	Unaffected
Addr. comparisons	Disabled	Unaffected
`memcpy`	Needs 16 b align	Unaffected
OS & Runtime Support		
Stack Allocations	Prot. by compiler	Android only
Global Variables	Prot. by linker	Android only
Anon. `mmap`	Supported.	Supported
File-backed `mmap`	Supported.	Not supported

Table 9: Comparison of CHERI and MTE features

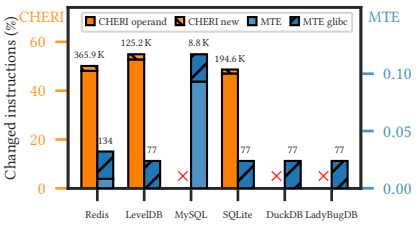


Figure 13: Changed instructions when compiling for MTE or CHERI.

	Instruction	Count
CHERI op	ldr	176,152 (26.7%)
	mov	100,409 (15.2%)
	str	68,118 (10.3%)
	add, ldp, adrp, stp, ldrb, …	315,358 (47.8%)
CHERI new	scbnds	16,827 (65.6%)
	clrperm	4,455 (17.4%)
	scvalue	1,942 (7.6%)
	seal, gcvalue, gctag, gclen, …	2,425 (9.5%)
MTE	irg	4,391 (47.9%)
	stg	4,386 (47.9%)
	ldg, st2g, stz2g, gmi, stzg	385 (4.2%)

Table 10: Instruction breakdown aggregated across evaluated systems.

using *uint64_t* or *uint128_t* discards capability metadata, leading to provenance issues. We replace them with *uintptr_t* and *intptr_t*, which preserve the capability metadata and the validity bit. Third, the *yyjson* library strips *const* qualifiers via a *size_t* cast, which drops capability bits; we replace it with the 128-bit *uintptr_t* in the CHERI version. Unfortunately, even after this effort, we only succeed in porting DuckDB to CHERI for simple queries.

For MTE, we port DuckDB's *jemalloc*, as performed for Redis. One additional issue arises: DuckDB stores cryptographic salts in the upper bits of pointers to protect hash tables against denial-of-service attacks. These bits overlap with the MTE tag field, corrupting either the tag or the salt. We patch DuckDB to place the salt in non-overlapping bits and mask out tag bits before hash calculations.

Results. We run the TPC-H workload [5] sweeping the scale factor from 1–100 and the thread count from 1–64. To measure the impact of IO operations on the performance overhead of MTE, we evaluate two configurations: *(i) cold*, where the OS caches are flushed before each query, and *(ii) warm*, where data resides already in the cache. As shown in Figure 12e, the geometric-mean overhead of MTE stays within 5% across all scale factors, thread counts, and both cache states, although individual queries deviate more strongly.

A single query, Q1, significantly deviates from the average overheads with 55% of overhead against a 2% median over the remaining queries. This overhead is due to the combination of the low aggregation cardinality and the number of rows processed, confirmed by profiling data: 97% of the additional cycles fall in DuckDB's aggregation methods. A low cardinality alone is not sufficient: Q8 and Q12 also group into two states, yet only add 3–4% overhead, as a join or a selective filter cuts their input first.

5.6 Graph Database System (LadyBugDB)

LadyBugDB is a graph database system, based on KuzuDB. It uses Compressed Sparse Row (CSR) indexing to store the graph topology, and uses analytical columnar format for node and edge properties. To avoid memory explosions and spills during graph traversals and joins, LadyBugDB leverages a factorized representation of intermediate results, keeping data compressed in memory.

Porting. For CHERI, we use the same CMake toolchain file that streamlines build system modifications and manually disable Scalable Vector Extension (SVE) support, since the Morello board does not implement it, but the build system fails to detect it automatically. Beyond these configuration changes, LadyBugDB runs without code modifications under the CHERI purecap architecture. For MTE, we instrument LadyBugDB's buffer manager's existing *getFrame()* function—which maps a frame or a page ID to a memory location—to tag the returned pointer before passing it to the rest of the system, making MTE fully transparent to the codebase.

Results. We evaluate LadyBugDB using queries from the official LDBC Social Network Benchmark (SNB), sweeping scale factors 10–100 and the thread count, which includes standard SNB queries and graph-specific operations (e.g., shortest-path search). As shown in Figure 12f, MTE introduces a geo-mean overhead of 9–14%, growing mildly with the thread count. In comparison, the mean overhead of CHERI is negligible across all configurations.

The outliers can be explained by the value types. Queries over variable-length string attributes, which are dictionary-encoded and must be materialized in heap-allocated memory, cost 10% overhead, against 4% for queries over fixed-width numeric attributes of the same table. 55% of the additional cycles are spent in the dictionary scan and a further 29% in the allocator, whose cost doubles because every allocation must have its region tagged and re-tagged on free.

Comparison with data structure analysis. Across all six systems, overheads are lower than for their underlying data structures (§4). The reason is that end-to-end execution also encompasses other tasks, including query parsing and planning, logging, *etc.*, which are typically less memory-intensive and amortize safety-check costs. System workloads exercise the whole database system, whereas our microbenchmarks isolate a single data structure operation.

Takeaway: Porting full database systems exposes the difference between MTE and CHERI in terms of portability and performance. MTE offers a highly portable integration path with minimal modifications to the allocator and maintains a low performance overhead of 5-10%. Conversely, CHERI demands deeper refactoring to correct memory alignments and capability provenance, which can hinder adoption if external dependencies are incompatible. Performance-wise, CHERI imposes a higher penalty, 20-60%, particularly on write-heavy key-value stores.

6 Compiler and Runtime Analysis

The performance and portability results (§4, §5) are not determined solely by the hardware mechanisms themselves. Several observed effects (e.g., disabled compiler optimizations, ABI incompatibilities) originate from the toolchain and OS layers above the hardware. Therefore, we analyze these layers to identify which constraints are fundamental to the hardware design and which are artifacts of current toolchain maturity. Table 9 summarizes the key findings. Overall, it shows a trade-off: CHERI requires disabling or constraining several pointer-related optimizations (aliasing, address comparisons, and `memcpy`), whereas MTE leaves these unaffected because it preserves the standard pointer representation. Conversely, CHERI

```
void foo(char *q, char *p) {
    if (q == p) { // `q == p` succeeds when called from bar
        *p = 0; // compiler assumes it may dereference `q`
    }
}
void bar(char *p) {
    foo(cheri_tag_clear(p), p); // same address, differing tag
}
```

Figure 14: Code being miscompiled by LLVM. By default, comparisons of capabilities ignore metadata.

benefits from support for protecting stack allocations, globals, and both anonymous and file-backed `mmap` regions, while MTE protection for stacks and globals is currently limited to Android and does not extend to file-backed `mmap`.

6.1 Compiler Analysis

For C and C++, *clang* [15] and *gcc* [11] forks exist for CHERI and Morello; MTE is also supported by both upstream compilers. Ongoing work extends CHERI support to other languages (i.e., Rust [59]).

Programming model. CHERI compilers support both *purecap* mode, where all pointers are capabilities, and *hybrid* mode, where only explicitly annotated pointers are. To manipulate capabilities, the compiler exposes intrinsics for operations such as setting or clearing the validity bit, adjusting bounds or permissions, and sealing/unsealing. In contrast, MTE requires no pointer annotation, preserving the type system and ABI. The compiler exposes intrinsics to manipulate logical tags and load or store allocation tags.

IR representation and optimizations. CHERI's LLVM fork represents capabilities using a separate address space, allowing them to coexist with normal pointers in hybrid code while preserving most optimization passes. Certain passes that assume 64-bit pointers must be disabled when compiling for capabilities. Further, at the time of writing, LLVM may miscompile capability comparisons by checking only addresses. Correct behavior requires treating capabilities with potentially differing bounds, permissions, or validity as distinct. Otherwise, optimizations may be missed or produce miscompilations (Figure 14). In this example, `foo` is called with two capabilities with the same address but differing tags. Since `q == p` only checks addresses, the compiler concludes that `q` and `p` are interchangeable and may dereference the untagged `q` in place of `p`. Another constraint appears when storing capabilities in untyped byte arrays. For small copies, the compiler may replace *memcpy* with a load-store sequence if 16-byte alignment can be proven statically. Code that stores capabilities in untyped byte buffers loses this guarantee, forcing a fallback to *memcpy* to preserve correctness.

Compiler-driven safety. Typically, compilers insert instructions to automatically provide safety when compiling to memory-safe hardware. For CHERI, the compiler transparently creates capabilities with appropriate bounds for stack allocations; the dynamic linker handles references to globals, whose addresses are unavailable at compile time. Subobject bounds narrowing (i.e., restricting a reference to a struct field or array element to its own bounds) is disabled by default for compatibility but is supported and can be enabled. This affects common database code patterns in which nodes store keys, payloads, and child pointers as subobjects. Our microbenchmarks, which create references to objects within an array, show no measurable performance difference between the two configurations; thus, enabling it globally is safe for most database code. Users can still opt out of subobject bounds narrowing for structs or fields where this leads to compatibility problems.

For MTE, LLVM provides a stack and globals sanitizer, while GCC support is under development. Rather than tagging each stack variable individually, which would require padding every object to the 16-byte tag granularity, the entire stack frame receives a single tag, preventing stack overflows. Correct stack and global tagging require support in the entire toolchain. The compiler must insert tagging and untagging code on function entries, returns, and when unwinding stack frames after non-local control flow (e.g., exceptions, *setjmp/longjmp* instructions), while the dynamic linker must be MTE-aware to enable MTE for stack and global regions. Currently, only the Android toolchain supports this end-to-end.

Figure 13 and Table 10 quantify the resulting code-generation changes across the evaluated systems. For CHERI, we see changes to instructions (e.g., *ldr*) which now operate on capabilities, as well as new instructions (e.g., *scbdns*) that manipulate capabilities. The large share of modified instructions means that the entire tooling stack, including debuggers and disassemblers, must be CHERI-aware to correctly interpret binaries. For MTE, in contrast, the binaries contain few explicit instructions; in several systems, most MTE-specific instructions reside in glibc rather than in the application. This highlights a key difference between the two mechanisms: CHERI changes pointer-carrying code throughout the binary, whereas MTE is limited to allocator and runtime support.

Takeaway: Toolchain maturity is a major factor in the porting complexity. MTE is highly portable by retaining existing ABIs and relies mostly on libraries. CHERI alters the entire binary, requiring specialized toolchains and increasing the porting effort.

6.2 Runtime Libraries

Hardware memory safety is only effective if the software that manages memory cooperates with it. Two components are central: the Application Binary Interface (ABI), the binary contract that fixes pointer size and layout across compiled modules, and the *memory allocator*, which hands out pointers (or capabilities). We discuss the constraints each mechanism places on these components.

ABI and binary compatibility. CHERI purecap binaries are ABI-incompatible with standard binaries. The generated ELF marks the ABI in a flag, and all linked libraries must be recompiled using the purecap ABI. As a partial solution, a hybrid ABI eases migration by allowing developers to introduce capabilities into unchanged code via pointer annotations. In contrast, MTE leaves the ABI unchanged, enabling adoption without recompiling dependencies.

Exception handling. Stack unwinding recognizes signal frames by reading instructions at the return address and matching them against the kernel's *rt_sigreturn* trampoline. Purecap forbids this because the program counter carries *execute* but not *load* permission, so inspecting code as data faults. Unwinders must instead obtain the bytes indirectly, *e.g.*, through a system call, which requires significant changes to the library implementation. As a result, we could not get exception handling working in our purecap builds. The database systems are unaffected, as they compile with *-fno-exceptions*.

Memory allocation. For heap protection, CHERI allocators must narrow the capabilities they return. Further, allocators keep book-

Figure 15: Memory overheads of MTE and CHERI.

keeping metadata adjacent to each allocation. When freeing memory, allocators cannot widen the user-facing, non-privileged pointer to access this metadata; instead, they must retain a separate, privileged capability for this. They must also pad allocations and avoid placing adjacent objects within loosely fitting bounds, as capability bounds are imprecise and may over-approximate, potentially leading to undetected illegal accesses.

For MTE, allocators must align allocations to 16-byte granules, generate tags, and assign them to memory. On Linux, MTE is limited to *mmap*-backed pages (the alternative to *sbrk*, which extends the heap contiguously), preventing allocators from placing objects in *sbrk*-based pages. Glibc's *ptmalloc* enables MTE through *GLIBC_TUNABLES*, assigning a random tag to the reserved region while leaving allocator metadata untagged. Other allocators may use deterministic schemes based on allocation classes. On free, memory is retagged with zero, so access through the old pointer triggers a mismatch and enables double-free detection.

Memory footprint overhead. We measure peak resident set size (RSS) under the workloads described in § 5 and report the normalized memory footprint relative to their baselines for MTE and CHERI in Figure 15. MTE imposes negligible overhead on systems only using custom allocators. However, systems that extensively use *malloc* for small allocations (e.g., MySQL) or the standard library containers to index the data (e.g., LadyBugDB) experience higher, though moderate, overhead from MTE's alignment and padding requirements. Conversely, CHERI's 128-bit capabilities heavily penalize systems with pointer-heavy data structures: Redis' and LevelDB's extensive use of hash tables and skip lists results in up to 66% memory overhead. In contrast, systems that maintain indexes using mostly integers, such as B+ trees in SQLite or CSR in LadyBugDB, experience negligible memory overhead with CHERI.

Temporal memory safety. CHERI can enforce temporal safety via capability revocation [45, 140, 143]. Freed regions are quarantined until a privileged capability revocation pass invalidates their capabilities, preventing *use-after-reallocation*. In contrast, MTE can retag freed memory, probabilistically invalidating existing pointers. As tags are finite, a reallocated region may receive the same tag as a dangling pointer, leaving stale accesses undetected. Allocators can reduce, but not eliminate, by delaying reuse through quarantine.

Takeaway: Hardware-assisted memory safety extensions place a significant performance and complexity burden on the memory allocator. CHERI inflates the memory footprint of pointer-heavy database systems by up to 66%. Similarly, MTE changes the complexity of memory allocations to $O(n)$, which can impact allocation-heavy database systems.

6.3 Operating System Support

In terms of operating system support, there are currently both FreeBSD [1] and Linux [2] forks providing CHERI support, while MTE is supported in mainline Linux since version 5.9.

Bug class	Baseline	MTE	CHERI
Heap buffer overflow	✗	✓ (into next object)	✓
Stack/global buffer overflow	✗	✗	✓
Out-of-bounds read/write	✗	✓ (89%)	✓
Integer overflow → corruption	libc	✓	✓
Use-after-free	✗	✓ (88%)	rev.
Double/invalid free	libc	✓ (double free only)	✓
Null pointer dereference	OS	OS	✓
Uninitialized memory access	✗	✗	✗

✓ detected ✓(…) partially detected libc allocator check
OS unmapped page ✗ undetected rev. needs revocation

Table 11: Detection of the memory-safety bug classes of our bug study (200 runs per test case).

Process initialization. For CHERI, the language runtime must constrain capabilities before transferring control to the program, including bounding the stack capability register and appropriately revoking permissions. For MTE, the process must call *prctl*, the process-control syscall, to enable tag checking. Individual memory regions must also enable MTE when they are mapped with *mmap* or when their protection is later changed with *mprotect*.

OS interfaces. CheriBSD [1] and Morello Linux [2] both expose capability and hybrid system-call interfaces; the hybrid is the default on Morello. Upon detecting an invalid access, both CHERI and MTE raise signals to the userspace process.

On Linux, MTE is restricted to anonymous memory mappings (memory not backed by a file, such as the heap) and must be explicitly enabled; file-backed mappings , which expose a file's contents as memory (as used for memory-mapped I/O), cannot be tagged. This restriction directly impacts database systems that rely on memory-mapped files for buffer management or storage access. If MTE safety is desired, data must be copied into an anonymous memory region, which may result in undesirable performance and memory penalties. Therefore, in our experiments (§ 5), MTE is limited to heap-allocated data and does not cover file-backed memory.

Takeaway: Although there is foundational OS support for MTE and CHERI, current implementations lack important features for database systems. Because Linux restricts MTE protection to anonymous memory, database systems cannot natively tag file-backed memory mappings. Meanwhile, enforcing a full pure-capability OS interface is challenging, leading CHERI-enabled OSes to default currently to a hybrid system-call interface.

6.4 Coverage of Memory-Safety Bug Classes

Finally, we evaluate how MTE and CHERI protect against the classes of bugs that we identified in our bug study § 2.1. We reproduce each class that our classifier considers memory-safety-related (Appendix B), *i.e.* including buffer overflows, out-of-bounds accesses, use-after-free, null pointer dereferences, as small test cases. We run each case 200 times without protection, with MTE, and with CHERI, and report in Table 11 whether each class is detected at the faulting access.

Spatial safety. CHERI deterministically detects heap, stack, and

global overflows and out-of-bounds accesses. In comparison, MTE protects only the heap, at the 16-byte granularity of its tags, and detects an access landing in another object only if both objects carry different tags. This behavior leads to probabilistic safety, where MTE catches 89% of the out-of-bounds accesses in our test scenario, due to tag collisions.

Temporal safety. MTE retags freed memory, so a use-after-free faults deterministically until the memory is reallocated and probabilistically afterwards. CHERI requires capability revocation to detect use-after-free, which CheriBSD supports but Morello Linux does not (§6.3). Both mechanisms catch double frees.

Null pointers and uninitialized memory. Without protection and with MTE, which only checks tags within tagged mappings, a null pointer dereference is stopped only because the OS leaves the first page unmapped. CHERI turns it into a capability fault at the faulting access: a null pointer carries no valid capability, so even accesses at large offsets from `NULL`, which may land in mapped memory without protection, are rejected. Uninitialized memory accesses are not detected by any mechanism, as neither tracks whether memory was written: a new allocation still exposes the previous owner's data. Allocators can close this gap by zeroing memory on allocation, for example by using special MTE instructions that tag and zero memory in a single operation, *i.e.* `stzg` (§3.2).

Takeaway: Without protection, the reproduced bugs silently corrupt or leak data, or are caught late by allocator checks, consistent with the crashes, incorrect results, and corruption of our bug study. MTE and CHERI instead stop most of these bug classes at the faulting access, before corrupted data propagates. CHERI deterministically enforces spatial safety but needs OS support for temporal safety, whereas MTE covers spatial and temporal heap bugs, but probabilistically and without protecting stack and global memory. Neither detects uninitialized memory accesses.

7 Related Work

Software-based approaches. Significant work has been done to retrofit memory safety into existing codebases written in unsafe languages. Trip-wire-based approaches pad allocations with inaccessible guard regions to catch out-of-bounds accesses [54, 120, 121]. Object-based approaches associate metadata, stored in disjoint tables or shadow memory, with an object, enforcing spatial safety via bounds checks on pointer manipulation and temporal safety by invalidating stale metadata [21, 27, 55, 60, 62, 85, 86, 105]. Pointer-based approaches embed metadata inline (e.g., fat pointers) or in separate memory regions, tracking all pointers to enforce temporal safety, and checking bounds on every access [40, 41, 73, 74, 102, 124, 146]; some approaches also combine per-pointer and per-object metadata [128, 139]. Orthogonal techniques include static analysis [39], parallelization of checks [150], and protection against memory error exploits via safe control-flow [17, 75] and software-based fault isolation [26, 58, 136, 147, 148]. Additionally, safe languages or dialects enforce memory safety at the type system [3, 67, 104]. Software approaches incur performance overheads that are prohibitive in production database deployments, a gap our work quantifies and motivates through hardware-assisted alternatives.

Hardware-based approaches. Hardware-based solutions reduce the overhead that makes software solutions impractical. Capability

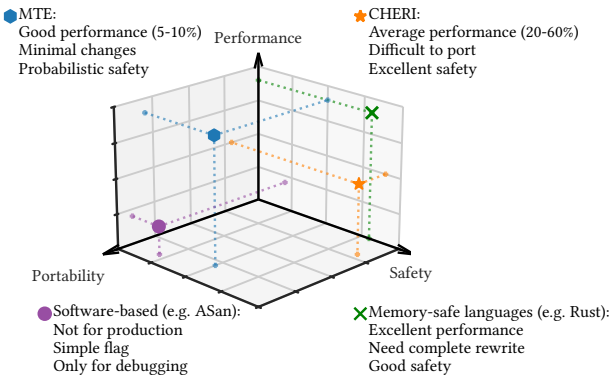


Figure 16: The database systems trilemma

architectures embed bounds and permissions in pointers. CHERI has been applied across a range of systems [22, 23, 45, 87, 137, 140, 143], extended to safe languages [59], while other alternatives have also been proposed [29, 80, 149]. Further works leverage MTE for deterministic memory tagging or isolation [56, 98]. Hardware accelerated bounds checking and pointer integrity have also been explored [38, 50, 70, 76, 82, 84, 100, 101, 131], as has hardware-based compartmentalization [46, 103, 144]. These works show the potential of hardware mechanisms but do not address database-specific challenges such as custom allocators. We provide the first side-by-side comparison of MTE and CHERI in this setting.

Hardware extension analysis. Prior surveys cover hardware improvements for memory-safe program execution [151] and the limitations and opportunities of modern isolation mechanisms [30]. Specifically, memory safety has been addressed through extensions such as Intel MPX [107] and Intel MPK [14, 110]. Adjacent work analyzes confidential computing hardware (e.g., Intel TDX, AMD SEV) [96] and confidential GPUs [63, 145]. Concurrent work analyzes the Arm MTE [69, 106, 132] and CHERI [72, 125] in isolation, treating database systems as black-box applications in benchmarks. Our work is complementary: rather than analyzing a single extension or a general-purpose workload, we conduct a *cross-layer* comparison of MTE and CHERI specifically for database systems, linking overheads to the data structures and microarchitectural access patterns.

Database testing. Efforts to improve database reliability have progressed from crash-oriented fuzzers [52, 119] to logic-bug detection, where a system returns incorrect results without crashing. Key advances include Pivoted Query Synthesis (PQS) [118] and Query Partitioning [117], popularized by SQLancer [36, 153]. Differential testing [68, 123] and reference engine construction approaches isolate optimizer and correctness bugs [116], while modern frameworks leverage coverage-guided techniques [48, 66, 138, 152] and relational constraint solvers [18, 19, 83] to explore complex query execution paths. Such approaches focus on detecting bugs after they occur, while we focus on *preventing* memory-safety exploitsand our bug study (§2.1) shows that such bugs exist even in well-tested systems. Unlike prior work on discovering database-system bugs [47], we classify existing reports and CVEs by their memory-safety implications, complementing recent LLM-based classification of security-relevant bugs and patches [20, 81, 130].

Takeaway	Implications
Architectural	
MTE tag must be loaded before a store	MTE loads have little overhead; stores are more expensive
CHERI capabilities double the memory to load on pointer access	CHERI memory accesses are more expensive than without capabilities
The hardware can prefetch tags/capabilities	Cache-aware data structures incur less performance overhead
CHERI imposes a large overhead for atomic operations on capabilities	Increases the performance tax on lock-free data structures
Concurrent modifications to tags/capabilities incur significant overheads	False sharing has more impact on memory-safe hardware
CHERI capabilities and MTE allocations must be aligned to 16 bytes	MTE and CHERI may increase memory fragmentation
Compiler/Runtime	
MTE requires minimal changes to the build toolchain and runtime	MTE is mostly compatible with existing software and programming tools
CHERI purecap requires deep modifications of core software components	CHERI development lacks support for many common programming tools
CHERI temporal safety requires additional revocation pass	Background revocation may create latency spikes in intensive workloads
MTE temporal safety faces the tag exhaustion challenge	Rapid memory reuse can lead to safety failures because of stale pointers
OSes provide exclusively hybrid or purecap APIs	Hybrid-API OSes offer flexibility at the cost of library dependence
MTE must be initialized exactly once per process	Host application must handle initialization for embedded database systems
MTE requires anonymous memory regions	Database systems cannot protect file-backed memory regions with MTE
Memory Allocation	
MTE/CHERI are supported by system allocators, *e.g.* glibc malloc	Database systems can benefit from safe allocations with minimal changes
Custom allocators must explicitly handle tag (MTE) or bounds (CHERI)	MTE/CHERI increase code complexity for custom allocations
MTE padding fragments memory; CHERI pointers bloat data structures	Increased memory footprints incur additional hardware costs
MTE allocation complexity is $O(n)$	Large, short-lived allocations should be protected by another mechanism
MTE-aware allocators must initialize tags	Allocators should use dedicated instructions for initialization (*stzg*/*stgp*).
Custom allocators typically maintain metadata for allocated blocks	CHERI allocators must sacrifice protection or maintain an additional pointer MTE tags can be loaded, simplifying the porting of legacy custom allocators
Database Systems	
Pointers carry address and metadata	Integer casts and metadata storage in unused pointer bits are disallowed
Ease of porting correlates with standard adherence	Legacy code assuming 64-bit pointers requires significant manual updates
Pointer indexes have higher overhead with CHERI than integer indexes	Indexing designers should consider the performance *vs.* safety tradeoff
Reads/scans incur relatively small overheads (MTE:2–6%; CHERI: 15–20%)	Read-heavy workloads are well-suited for early hardware safety adoption
Inserts lead to larger overheads (MTE: 20–40%; CHERI: up to 200%)	Reduces the advantage of in-memory systems for write-heavy workloads
Overheads of OLTP workloads are lower than OLAP and KVS workloads	Memory-bound workloads face the highest performance tax

Table 12: CHERI and MTE analysis takeaways and their implications for database systems practitioners.

8 The Database Systems Trilemma

This paper presents the first side-by-side, cross-layer comparison of MTE and CHERI on a database workload suite, spanning from microarchitecture through an indexed-data-structure layer to full systems. We investigate the answers that MTE and CHERI offer to the memory safety trilemma (§ 2.2) across micro-architectural analysis, data structures, and full-fledged database systems. We summarize our findings in Table 12 and compare against software-based approaches and memory-safe languages in Figure 16.

The two extensions take different approaches. CHERI provides strong, deterministic guarantees by enforcing pointer bounds, permissions, and unforgeability. It detects spatial violations, including out-of-bounds accesses and buffer overflows, provides temporal safety through capability revocation, and catches intra-object overflows when the compiler narrows sub-object bounds. By contrast, MTE detects the same bug classes probabilistically and cannot detect intra-object overflows. Neither mechanism catches type confusion, valid-but-incorrect pointer use, or other logic bugs.

This divergence inversely correlates with performance: CHERI imposes significant overheads (20–60%) across database system workloads, whereas MTE's average overhead is moderate (5–10%). Finally, the invasiveness of the two solutions dictates their practical portability. MTE requires minimal changes to the build and deployment pipeline because it is fully supported by current toolchains. On the other hand, the CHERI toolchain introduces considerable complexity due to the fundamental change in pointer size. Its purecap ABI also requires compatible builds of all dependent libraries, which demands significant engineering effort for use in large, dependency-heavy applications like database systems. Overall, MTE and CHERI show that the Pareto frontier has moved beyond what software-based approaches can offer, making hardware enforcement a viable path to production-grade memory safety in database systems.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. We also thank Viktor Leis for his feedback on the project. This work was supported in part by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), the Intel Trustworthy Data Center of the Future (TDCoF), Google Research Grants, and the TUM Innovation Network Resilient, Trustworthy, Sustainable (ReTruSt).

References

[1] [n. d.]. CheriBSD. https://www.cheribsd.org/
[2] [n. d.]. Morello Linux. https://linux.morello-project.org/
[3] [n. d.]. Rust Programming Language. https://rust-lang.org/
[4] [n. d.]. TPC-C. https://www.tpc.org/tpcc/
[5] [n. d.]. TPC-H. https://www.tpc.org/tpch/
[6] 2015. Hardware-Assisted Checking Using Silicon Secured Memory (SSM). https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html. Accessed: 2026-03-04.
[7] 2015. Poject Zero - Stagefrightened? https://googleprojectzero.blogspot.com/2015/09/stagefrightened.html. Accessed: 2026-03-04.
[8] 2019. A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. Accessed: 2026-03-04.

[9] 2019. Queue the Hardening Enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html. Accessed: 2026-03-04.
[10] 2020. The Heartbleed Bug. https://heartbleed.com/. Accessed: 2026-03-04.
[11] 2023. Morello Gcc. https://git.morello-project.org/morello/gnu-toolchain/gcc
[12] 2025. Arm memory tagging extension. https://source.android.com/docs/security/test/memory-safety/arm-mte. Accessed: 2026-03-04.
[13] 2025. Intel(R) Memory Protection Extensions (MPX). https://www.kernel.org/doc/html/v5.2/x86/intel_mpx.html. Accessed: 2026-03-04.
[14] 2025. Memory Protection Keys. https://docs.kernel.org/core-api/protection-keys.html. Accessed: 2026-03-04.
[15] 2025. Morello Llvm-Project. https://git.morello-project.org/morello/llvm-project
[16] 2025. Morello Musl Libc. https://git.morello-project.org/morello/musl-libc
[17] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (Nov. 2009), 4:1–4:40. doi:10.1145/1609956.1609960
[18] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 238–247. doi:10.1109/ASE.2008.34
[19] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL query generation for systematic testing of database engines. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 329–332. ISBN 9781450301169. doi:10.1145/1858996.1859063
[20] Md Basim Uddin Ahmed, Nima Shiri Harzevili, Jiho Shin, Hung Viet Pham, and Song Wang. 2025. SecVulEval: Benchmarking LLMs for Real-World C/C++ Vulnerability Detection. arXiv:2505.19828 [cs.SE] doi:10.48550/arXiv.2505.19828
[21] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, USA, 51–66.
[22] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W. Moore, Robert Norton-Wright, Margo Seltzer, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2025. CHERIoT RTOS: An OS for Fine-Grained Memory-Safe Compartments on Low-Cost Embedded Devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 67–84. ISBN 979-8-4007-1870-0. doi:10.1145/3731569.3764844
[23] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Toronto ON Canada, 641–653. ISBN 979-8-4007-0329-4. doi:10.1145/3613424.3614266
[24] Amazon. 2026. Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/mgmt/welcome.html.
[25] Android Open Source Project. 2013. Arm Memory Tagging Extension. https://source.android.com/docs/security/test/memory-safety/arm-mte. Accessed: 2026-03-06.
[26] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. 2011. Language-Independent Sandboxing of Just-in-Time Compilation and Self-Modifying Code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 355–366. ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993540
[27] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. 2022. SafePM: A Sanitizer for Persistent Memory. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 506–524. ISBN 978-1-4503-9162-7. doi:10.1145/3492321.3519574
[28] Adam Byerly and Daniel Khashabi. 2026. Self-Consistency Falls Short! The Adverse Effects of Positional Bias on Long-Context Problems. *Transactions of the Association for Computational Linguistics* 14 (April 2026), 292–317. doi:10.1162/TACL.a.625
[29] Tony Chen, David Chisnall, and David Chisnall. 2025. CHERI-Lite for Memory Safety. (2025).
[30] Xiangdong Chen, Zhaofeng Li, Tirth Jain, Vikram Narayanan, and Anton Burtsev. 2024. Limitations and Opportunities of Modern Hardware Isolation Mechanisms. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 349–368. ISBN 978-1-939133-41-0. https://www.usenix.org/conference/atc24/presentation/chen-xiangdong
[31] ClickHouse, Inc. 2026. ClickHouse: A Fast Open-Source OLAP Database Management System. https://github.com/ClickHouse/ClickHouse.
[32] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. ISBN 978-1-4503-0036-0. doi:10.1145/1807128.1807152
[33] Intel Corporation. 2025. *ChkTag: x86 Memory Safety*. https://community.intel.com/t5/Blogs/Tech-Innovation/open-intel/ChkTag-x86-Memory-Safety/post/1721490 Accessed: 2026-03-10.
[34] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 631–644. doi:10.1145/2786763.2694359
[35] Turso DB. [n. d.]. Turso, an in-process SQL database, compatible with SQLite. Retrieved 2026-02-06 from https://github.com/tursodatabase/turso
[36] SQLancer Developers. [n. d.]. SQLancer. Retrieved 2026-02-05 from https://github.com/sqlancer/sqlancer
[37] SQLite Developers. [n. d.]. SQLite Home Page. Retrieved 2024-07-24 from https://www.sqlite.org/
[38] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 103–114. ISBN 9781595939586. doi:10.1145/1346281.1346295
[39] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 144–157. ISBN 978-1-59593-320-1. doi:10.1145/1133981.1133999
[40] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. Association for Computing Machinery, New York, NY, USA, 132–142. ISBN 978-1-4503-4241-4. doi:10.1145/2892208.2892212
[41] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. ISBN 978-1-891562-46-4. doi:10.14722/ndss.2017.23287
[42] DuckDB Labs. 2026. DuckDB: An Embeddable Analytical Database. https://github.com/duckdb/duckdb.
[43] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the impact of memory allocation on high-performance query processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–3.
[44] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. KÙZU Graph Database Management System. In *Conference on Innovative Data Systems Research (CIDR)*, Vol. 7. 25–35.
[45] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2024. Cornucopia Reloaded: Load Barriers for CHERI Heap Temporal Safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24, Vol. 2)*. Association for Computing Machinery, New York, NY, USA, 251–268. ISBN 979-8-4007-0385-0. doi:10.1145/3620665.3640416
[46] Martin Fink, Dimitrios Stavrakakis, Dennis Sprokholt, Soham Chakraborty, Jan-Erik Ekberg, and Pramod Bhatotia. 2025. Cage: Hardware-Accelerated Safe WebAssembly. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 538–552. ISBN 9798400712753. doi:10.1145/3696443.3708920
[47] Xiyue Gao, Zhuang Liu, Jiangtao Cui, Hui Li, Hui Zhang, Kewei Wei, and Kankan Zhao. 2023. A Comprehensive Survey on Database Management System Fuzzing: Techniques, Taxonomy and Experimental Comparison. arXiv:2311.06728 [cs.DB] doi:10.48550/arXiv.2311.06728
[48] Xiyue Gao, Zhuang Liu, Yiran Shen, Hui Li, Yingfan Liu, Hongjun Xiao, Yanguo Peng, and Jiangtao Cui. 2025. Fucci: Database Transaction Fuzzing via Random Conflict Construction and Multilevel Constraint Solving. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1879–1891. doi:10.14778/3725688.3725713
[49] ggml. 2026. Llama.cpp. https://github.com/ggml-org/llama.cpp. Last accessed on 2026-03-02.
[50] Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. 2009. Architectural support for low overhead detection of memory violations. In *2009 Design, Automation & Test in Europe Conference & Exhibition*. 652–657. doi:10.1109/DATE.2009.5090747
[51] Google. 2019. Adopting the Arm Memory Tagging Extension in Android. https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html. Accessed: 2026-03-06.
[52] Google. [n. d.]. AFL: american fuzzy lop. Retrieved 2026-02-05 from https://github.com/google/AFL
[53] Google LLC. 2026. LevelDB: A Fast Key-Value Storage Library. https://github.com/google/leveldb.

[54] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. 2023. TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 535–552. ISBN 978-1-939133-34-2. https://www.usenix.org/conference/osdi23/presentation/gopal

[55] Floris Gorter and Cristiano Giuffrida. 2025. RangeSanitizer: detecting memory errors with efficient range checks. In *Proceedings of the 34th USENIX Conference on Security Symposium* (Seattle, WA, USA) *(SEC '25)*. USENIX Association, USA, Article 232, 19 pages. ISBN 978-1-939133-52-6.

[56] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. 2024. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4239–4257. doi:10.1109/SP54263.2024.00263

[57] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (May 2023), 50–57. doi:10.1109/MM.2023.3264676

[58] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. doi:10.1145/3140587.3062363

[59] Sarah Harris, Simon Cooksey, Michael Vollmer, and Mark Batty. 2023. Rust for Morello: Always-On Memory Safety, Even in Unsafe Code. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 39:1–39:27. ISBN 978-3-95977-281-5. doi:10.4230/LIPIcs.ECOOP.2023.39

[60] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-Weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. Association for Computing Machinery, New York, NY, USA, 135–144. ISBN 978-1-4503-1206-6. doi:10.1145/2259016.2259034

[61] Red Hat. 2015. GHOST: glibc vulnerability (CVE-2015-0235). https://access.redhat.com/articles/1332213. Accessed: 2026-03-04.

[62] Junwha Hong, Wonil Jang, Mijung Kim, Lei Yu, Yonghwi Kwon, and Yuseok Jeon. 2025. CMASan: Custom Memory Allocator-Aware Address Sanitizer. In *2025 IEEE Symposium on Security and Privacy (SP)*. 740–757. doi:10.1109/SP61157.2025.00110

[63] Antonio Martínez Ibarra, Julian James Stephen, Aurora González Vidal, K. R. Jayaram, and Antonio Fernando Skarmeta Gómez. 2025. Performance of Confidential Computing GPUs. arXiv:2505.16501 [cs.PF] https://arxiv.org/abs/2505.16501

[64] Google Inc. 2026. LevelDB. Google. https://github.com/google/leveldb

[65] influxdata. 2026. InfluxDB. https://github.com/influxdata/influxdb.

[66] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4949–4965. ISBN 978-1-939133-37-3. https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming

[67] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, USA, 275–288. ISBN 978-1-880446-00-3.

[68] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.* 13, 1 (Sept. 2019), 57–70. doi:10.14778/3357377.3357382

[69] Shiv Kaushik, Mahesh Madhav, Nagi Aboulenein, Jason Bessette, Sandeep Brahmadathan, Ben Chaffin, Matthew Erler, Stephan Jourdan, Thomas Maciukenas, Ramya Masti, Jon Perry, Massimo Sutera, Scott Tetrick, Bret Toll, David Turley, Carl Worth, and Atiq Bajwa. 2025. Optimized Memory Tagging on AmpereOne Processors. arXiv:2511.17773 [cs] doi:10.48550/arXiv.2511.17773

[70] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based Always-On Heap Memory Safety. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1153–1166. doi:10.1109/MICRO50266.2020.00095

[71] Alexey Kopytov. 2026. Sysbench. https://github.com/akopytov/sysbench

[72] John Alistair Kressel, Hugo Lefeuvre, and Pierre Olivier. 2023. Software Compartmentalization Trade-Offs with Hardware Capabilities. arXiv:2309.11332 [cs.CR] doi:10.48550/arXiv.2309.11332

[73] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks without the Checks. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, 1–14. ISBN 978-1-4503-5584-1. doi:10.1145/3190508.3190553

[74] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 205–221. ISBN 978-1-4503-4938-3. doi:10.1145/3064176.3064192

[75] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 147–163. ISBN 978-1-931971-16-4.

[76] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. 2013. 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 (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 721–732. ISBN 978-1-4503-2477-9. doi:10.1145/2508859.2516713

[77] LadybugDB. [n. d.]. LadybugDB. LadybugDB. https://github.com/LadybugDB/ladybug

[78] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (May 2023), 7:1–7:25. doi:10.1145/3588687

[79] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. doi:10.1109/ICDE.2013.6544812

[80] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 253–267. ISBN 9781450385572. doi:10.1145/3466752.3480076

[81] Xingyu Li, Juefei Pu, Yifan Wu, Xiaochen Zou, Shitong Zhu, Qiushi Wu, Zheng Zhang, Joshua Hsu, Yue Dong, Zhiyun Qian, Kangjie Lu, Trent Jaeger, Michael De Lucia, Srikanth V. Krishnamurthy, Shitong Zhu, Qiushi Wu, and Joshua Hsu. 2026. What Do They Fix? LLM-Aided Categorization of Security Patches for Critical Memory Bugs. In *Proceedings 2026 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. ISBN 979-8-9919276-8-0. doi:10.14722/ndss.2026.230328

[82] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1901–1915. ISBN 9781450394505. doi:10.1145/3548606.3560598

[83] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326. ISBN 978-1-939133-31-1. https://www.usenix.org/conference/usenixsecurity22/presentation/liang

[84] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC It up: Towards Pointer Integrity Using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, USA, 177–194. ISBN 978-1-939133-06-9.

[85] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. 2024. CAMP: Compiler and Allocator-based Heap Memory Protection. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/conference/usenixsecurity24/presentation/lin

[86] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. 2024. GiantSan: Efficient Memory Sanitization with Segment Folding. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2024)*.

[87] Duncan Lowther, Dejice Jacob, and Jeremy Singer. 2023. CHERI Performance Enhancement for a Bytecode Interpreter. arXiv:2308.05076 [cs] http://arxiv.org/abs/2308.05076

[88] Arm Ltd. 2019. *Armv8.5-A Memory Tagging Extension*. Technical Report. https://developer.arm.com/documentation/102925/latest

[89] Arm Ltd. 2022. *Arm Architecture Reference Manual Supplement - Morello for A-profile Architecture*. Technical Report. https://developer.arm.com/documentation/ddi0606/latest

[90] Arm Ltd. 2025. *Arm® Architecture Reference Manual for A-profile Architecture*. Technical Report. https://developer.arm.com/documentation/ddi0487/latest

[91] MariaDB Foundation. 2026. MariaDB Server. https://github.com/MariaDB/server.

[92] Ilya Meignan--Masson, Martin Fink, Masanori Misono, Dimitrios Stavrakakis, and Pramod Bhatotia. 2026. Should Your Database Systems Use Hardware-Assisted Memory Safety Extensions in Production? *Proc. ACM Manag. Data* 4, 6 (SIGMOD), Article 378 (Dec. 2026). doi:10.1145/3856373

[93] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert NM Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. *ACM SIGPLAN Notices* 51, 6 (2016), 1–15.

[94] memcached. 2026. memcached: a high performance multithreaded event-based key/value cache store. https://github.com/memcached/memcached.

[95] Meta Platforms, Inc. 2026. RocksDB: A Persistent Key-Value Store for Flash and

RAM Storage. https://github.com/facebook/rocksdb.
[96] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. 2025. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *SIGMETRICS Perform. Eval. Rev.* 53, 1 (June 2025), 34–36. doi:10.1145/3744970.3727280
[97] MITRE Corporation. 2025. CVEProject/cvelistV5: Official CVE List in JSON 5.0 Format. https://github.com/CVEProject/cvelistV5.
[98] Marius Momeu, Alexander J. Gaidis, Jasper v.d. Heidt, and Vasileios P. Kemerlis. 2025. IUBIK: Isolating User Bytes in Commodity Operating System Kernels via Memory Tagging Extensions. In *2025 IEEE Symposium on Security and Privacy (SP)*. 867–885. doi:10.1109/SP61157.2025.00135
[99] MongoDB, Inc. 2026. MongoDB: A Document-Oriented Database. https://github.com/mongodb/mongo.
[100] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 189–200. doi:10.1109/ISCA.2012.6237017
[101] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 175–184. ISBN 9781450326704. doi:10.1145/2581122.2544147
[102] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. *SIGPLAN Not.* 44, 6 (June 2009), 245–258. doi:10.1145/1543135.1542504
[103] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 266–281. ISBN 978-1-4503-9918-0. doi:10.1145/3582016.3582023
[104] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. doi:10.1145/1065887.1065892
[105] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 42, 6 (June 2007), 89–100. doi:10.1145/1273442.1250746
[106] Taehyun Noh, Yingchen Wang, Tal Garfinkel, Mahesh Madhav, Daniel Moghimi, Mattan Erez, and Shravan Narayan. 2026. ARM MTE Performance in Practice (Extended Version). arXiv:2601.11786 [cs.CR] https://arxiv.org/abs/2601.11786
[107] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2 (June 2018), 28:1–28:30. doi:10.1145/3224423
[108] Inc. Oracle. [n. d.]. MySQL. https://www.mysql.com/
[109] Oracle Corporation. 2026. MySQL Server. https://github.com/mysql/mysql-server.
[110] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, USA, 241–254. ISBN 978-1-939133-03-8.
[111] Andy Pavlo. 2025. Py-Tpcc. https://github.com/apavlo/py-tpcc
[112] The Chromium Project. [n. d.]. Memory Safety. https://www.chromium.org/Home/chromium-security/memory-safety/
[113] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. ISBN 978-1-4503-5643-5. doi:10.1145/3299869.3320212
[114] Redis. [n. d.]. Redis - The Real-time Data Platform. https://redis.io/
[115] Redis Ltd. 2026. Redis: In-Memory Data Structure Store. https://github.com/redis/redis.
[116] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1140–1152. ISBN 9781450370431. doi:10.1145/3368089.3409710
[117] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (Nov. 2020), 30 pages. doi:10.1145/3428279
[118] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 667–682. ISBN 978-1-939133-19-9. https://www.usenix.org/conference/osdi20/presentation/rigger
[119] Andreas Seltenreich. [n. d.]. SQLsmith. Retrieved 2026-02-05 from https://github.com/anse1/sqlsmith
[120] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. ISBN 978-931971-93-5. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany
[121] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2024. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 168–177. ISBN 979-8-4007-0501-4. doi:10.1145/3639477.3640328
[122] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and How It Improves C/C++ Memory Safety. arXiv:1802.09517 [cs] doi:10.48550/arXiv.1802.09517
[123] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622. ISBN 1558605665.
[124] Dimitrios Stavrakakis, Alexandrina Panfil, MJin Nam, and Pramod Bhatotia. 2024. SPP: Safe Persistent Pointers for Memory Safety. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 37–52. doi:10.1109/DSN58291.2024.00019
[125] Xiaoyang Sun, Jeremy Singer, and Zheng Wang. 2025. Sweet or Sour CHERI: Performance Characterization of the Arm Morello Platform. In *2025 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 423–438. ISBN 979-8-3315-4917-6. doi:10.1109/IISWC66894.2025.00042
[126] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.
[127] Rustam Talibzade, Idilio Drago, and Francesco Bergadano. 2025. On Using LLMs for Vulnerability Classification. In *Proceedings of the 2025 Workshop on Large AI Systems and Models with Privacy and Security Analysis (LAMPS '25)*. Association for Computing Machinery, New York, NY, USA, 68–71. ISBN 979-8-4007-1896-0. doi:10.1145/3733800.3763267
[128] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proc. ACM Program. Lang.* 7, PLDI, Article 111 (June 2023), 24 pages. doi:10.1145/3591225
[129] The PostgreSQL Global Development Group. 2026. PostgreSQL. https://www.postgresql.org/.
[130] Mohammad Jalili Torkamani, Joey NG, Nikita Mehrotra, Mahinthan Chandramohan, Padmanabhan Krishnan, and Rahul Purandare. 2025. Streamlining Security Vulnerability Triage with Large Language Models. arXiv:2501.18908 [cs.SE] doi:10.48550/arXiv.2501.18908
[131] Martin Unterguggenberger, David Schrammel, Lukas Lamster, Pascal Nasahl, and Stefan Mangard. 2023. Cryptographically Enforced Memory Safety. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (<conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-loc>) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 889–903. ISBN 9798400700507. doi:10.1145/3576915.3623138
[132] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. 2023. Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia) *(ASIA CCS '23)*. Association for Computing Machinery, New York, NY, USA, 177–189. ISBN 9798400700989. doi:10.1145/3579856.3590331
[133] Victor van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future, Vol. 7462. ISBN 978-3-642-33337-8. doi:10.1007/978-3-642-33338-5_5
[134] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 86–106.
[135] Jeff Vander Stoep and Alex Rebert. 2024. Google Online Security Blog: Eliminating Memory Safety Vulnerabilities at the Source. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html
[136] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. ISBN 978-0-89791-632-5. doi:10.1145/168619.168635
[137] Kui Wang, Dmitry Kasatkin, Vincent Ahlrichs, Lukas Auer, Konrad Hohentanner, Julian Horsch, and Jan-Erik Ekberg. 2024. Cherifying Linux: A Practical View on Using CHERI. In *Proceedings of the 17th European Workshop on Systems Security (EuroSec '24)*. Association for Computing Machinery, New York, NY, USA, 15–21. ISBN 979-8-4007-0542-7. doi:10.1145/3642974.3652282
[138] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level

DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 328–337. doi:10.1109/ICSE-SEIP52600.2021.00042

[139] Xiaolei Wang, Bin Zhang, Chaojing Tang, and Long Zhang. 2024. Highly Comprehensive and Efficient Memory Safety Enforcement with Pointer Tagging. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 74–81. doi:10.1109/DSN-W60302.2024.00026

[140] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. 608–625. doi:10.1109/SP40000.2020.00098

[141] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 457–468. doi:10.1145/2678373.2665740

[142] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 457–468. doi:10.1145/2678373.2665740

[143] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 545–557. ISBN 978-1-4503-6938-1. doi:10.1145/3352460.3358288

[144] Jiali Xu, Mengyao Xie, Chenggang Wu, Yinqian Zhang, Qijing Li, Xuan Huang, Yuanming Lai, Yan Kang, Wei Wang, Qiang Wei, and Zhe Wang. 2023. PANIC: PAN-assisted Intra-process Memory Isolation on ARM. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Copenhagen Denmark, 919–933. ISBN 979-8-4007-0050-7. doi:10.1145/3576915.3623206

[145] Yang Yang, Mohammad Sonji, and Adwait Jog. 2025. Dissecting Performance Overheads of Confidential Computing on GPU-based Systems. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 1–16. doi:10.1109/ISPASS64960.2025.00020

[146] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 88–99. doi:10.1109/ISSRE.2014.20

[147] Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, La Jolla CA USA, 649–665. ISBN 979-8-4007-0385-0. doi:10.1145/3620665.3640408

[148] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: A Sandbox for Portable, Untrusted X86 Native Code. *Commun. ACM* 53, 1 (Jan. 2010), 91–99. doi:10.1145/1629175.1629203

[149] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 787–804. ISBN 978-1-939133-37-3. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason

[150] Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuandong Li, and Zhiqiang Zuo. 2023. Catamaran: Low-Overhead Memory Safety Enforcement via Parallel Acceleration. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 816–828. ISBN 9798400702211. doi:10.1145/3597926.3598098

[151] Lianying Zhao, He Shuang, Shengjie Xu, Wei Huang, Rongzhen Cui, Pushkar Bettadpur, and David Lie. 2024. A Survey of Hardware Improvements to Secure Program Execution. *ACM Comput. Surv.* 56, 12, Article 311 (Oct. 2024), 37 pages. doi:10.1145/3672392

[152] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. ISBN 9781450370899. doi:10.1145/3372297.3417260

[153] Suyang Zhong and Manuel Rigger. 2026. Scaling Automated Database System Testing. arXiv:2503.21424 [cs.SE] https://arxiv.org/abs/2503.21424

Table 13: Experimental platform specifications.

	Morello	Ampere 1a	Xeon 6438Y+
Extension	CHERI	MTE	MPK
ISA	Armv8.2	Armv8.6	x86-64
Microarch.	Neoverse N1	AmpereOne	Golden Cove
Sockets × Cores	1 × 4	1 × 192	2 × 32
Frequency	2.5 GHz	3.2 GHz	2.0 GHz
L1 I-Cache	64 KiB	16 KiB	32 KiB
L1 D-Cache	64 KiB	64 KiB	48 KiB
L2 Cache	1 MiB	2 MiB	2 MiB
L3 Cache	1 MiB (shared/2 cores)	—	60 MiB (shared/socket)
RAM	16 GiB	512 GiB	2 TiB
OS	Debian 13.2	NixOS 25.11	NixOS 25.11
Kernel	Morello Linux 6.7.0	Linux 6.17.13	Linux 6.0.0

All cache sizes are per core unless noted otherwise.

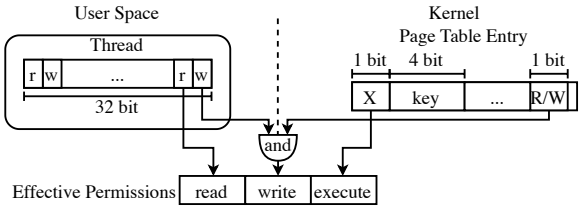


Figure 17: MPK access check mechanism.

A Additional Architectural Analysis

A.1 MPK

Experimental Setup. We evaluate Memory Protection Keys (MPK) on a dual-socket Intel Xeon Gold 6438Y+ CPU with 32 cores per socket and 2 threads per core, featuring 2 TiB of memory, running NixOS 25.05 and Linux 6.0.0. Each core features 32 KiB of L1i cache, 48 KiB of L1d cache, 2 MiB of L2 cache, while each core additionally features 60 MiB of L3 cache.

Mechanism. MPK introduces a per-thread isolation mechanism on a page-level granularity. Four unused bits in the Page Table Entry (PTE) specify one of 16 distinct protection keys. Each thread then specifies its access rights in the thread-local PKRU register, which holds two bits for each key, specifying read and write permissions. This allows for very fast switching of access rights, compared to *mprotect* system calls. Code using MPK allocates a new protection key by calling *pkey_alloc*. *pkey_mprotect* associates a protection key with a memory region, while *pkey_free* marks the key as free again. To mirror *mprotect*'s model, user-space code must synchronize the PKRU register between threads [110]. MPK *complements mprotect*'s permissions: With *mprotect*, users can select between five permissions: inaccessible, read-only, read-execute, read-write, and read-write-execute. Execute-only is not possible. Conversely, MPK allows to enable read or write permissions. The effective permission is that of the intersection of the page table entry and the PKRU register. Using MPK allows to set a page to execute-only, by disallowing reading through MPK.

However, MPK imposes some drawbacks: As the number of keys is limited to 16, allocating new keys may fail. Freeing keys does not invalidate existing keys, allowing for key use-after-free scenarios where pages using the freed are accessible using the reallocated key. Both of these problems can be solved by virtualizing keys in software, e.g., through libmpk [110]. Since the protection key is set per page, protection is more coarse-grained than CHERI or MTE: Providing fine-grained memory safety by protecting individual objects is not feasible. Instead, MPK is used to divide sensitive and non-sensitive data (e.g., shadow stacks), isolate different compartments (e.g., JIT-compiled functions), or allow for execute-only permissions.

Table 14: MPK Instructions

Inst	Description	Cycles (↓)
rdpkru	Read protection key rights.	4.02
wrpkru	Writes protection key rights.	17.78
pkru update	Read, change and write protection key rights.	18.66
System Call	**Description**	**Cycles (↓)**
pkey_alloc	Allocate a new protection key.	272.84
pkey_free	Free a protection key.	268.36
pkey_mprotect	Change protection key for region.	447.75

↓: lower is better

Microarchitectural Analysis. We evaluate the overheads of MPK, in particular the cost of reading, writing, and updating the PKRU register, as well as the cost of the newly introduced system calls in Table 14. As accessing the PKRU register is thread-local and done in user space, it performs very well. Updating the permissions completes in under 19 cycles. The system calls are much more expensive, taking around 270 cycles to allocate or free a protection key and around 450 cycles to change the protection key of a memory page.

B Memory Safety Bug Study

This section describes the full methodology used to collect, classify, and analyze memory safety bugs across the database systems studied in §2.1.

Scope and Time Window. The study covers two complementary datasets with overlapping system coverage.

Our *bug report study* covers nine database systems:

- DuckDB [42]
- Redis [115]
- RocksDB [95]
- ClickHouse [31]
- MariaDB [91]
- MySQL [109]
- LevelDB [53]

Our *CVE study* covers thirteen database systems:

- MySQL [109]
- SQLite [37]
- MariaDB [91]
- Redis [115]
- LevelDB [53]
- RocksDB [95]
- DuckDB [42]
- LadyBug [77]
- PostgreSQL [129]
- MongoDB [99]
- Memcached [94]
- InfluxDB [65]
- Redshift [24]

All data collection is restricted to the ten-year window *2015–2025* to ensure coverage of modern vulnerability trends.

B.1 CVE Collection

We download the full *CVEProject/cvelistV5* snapshot [97][2], which provides every CVE in structured JSON format following the CVE 5.0

[2] https://github.com/CVEProject/cvelistV5

schema. To isolate the relevant CVEs for the systems considered in this study, we scan the *containers.cna.affected* fields (*product* and *packageName*) for case-insensitive matches against the database name. We did not include the CVE *description* in our pattern-matching search because we observed that it returned various CVEs unrelated to the database system itself. CVE records whose filenames encode a year prior to 2015 are skipped at the filename level before parsing. Results are filtered by the *cveMetadata.datePublished* field to enforce the 2015-2025 window. The output is a per-database list of matching CVE files that are fed into the classification step.

B.2 Bug Report Collection

Since CVEs capture only publicly disclosed, security-relevant vulnerabilities, we expand our study to include a dataset with bug reports scraped from each project's official issue tracker. All scrapers impose a 0.5 s inter-request delay to respect rate limits of the bug tracker hosts. We save each bug as a self-contained JSON file.

GitHub Issues. For *DuckDB, RocksDB, Redis, LevelDB, and ClickHouse*, we query the GitHub REST API (*/issues*) with *state=closed*, sorted ascending by *updated_at*, and pre-filtered with the API's *since* parameter set to *2015-01-01T00:00:00Z*. To work around GitHub's 10,000-result depth limit per sorted query, the scraper slides a *since* cursor forward to the latest *updated_at* timestamp seen whenever the page counter approaches the depth limit (page $\geq$ 90), effectively re-anchoring the query window. Pull requests are excluded, and results are post-filtered to enforce the 2015-2025 window.

Jira. For MariaDB[3], we query each project's public Jira REST API (*/rest/api/2/search*) using JQL date constraints (*created >= "2015-01-01" AND created <= "2025-12-31"*), and paginate through all results in batches of 50. Issue comments are concatenated to the description to provide a full context to the classifier.

MySQL Bug Tracker. For MySQL, we use the CSV export endpoint of the official MySQL bug tracker[4]. We set the *bug_age* parameter to the number of days elapsed since *2015-01-01*, pre-filtering server-side, and post-filter individual rows against the target window using the *Entered* timestamp in the CSV. For each bug ID, the full HTML report page (*bugs.mysql.com/bug.php?id={ID}*) is fetched and parsed to extract the title, status, and comment text.

B.3 Keyword Filtering

Before invoking the LLM classifier, we apply a case-insensitive regular-expression search over each bug's full text (title, description, and comments) using the following list of memory-safety terms:

> buffer overflow/overrun/underflow/underrun, out-of-bounds (OOB, OOBR, OOBW), array bounds, off-by-one, stack overflow, heap overflow, stack smashing, integer overflow/underflow, null pointer dereference, double free, multiple free, free-after-free, use-after-free (UAF), stack/heap use-after, uninitialized memory/variable, wild pointer, dangling pointer/reference, memory safety/corruption, heap/stack corruption, invalid/illegal memory access, invalid read/write, segmentation fault (SIGSEGV), address sanitizer, ASLR

Bugs that match at least one term form a *keyword set* that serves as a baseline against which LLM results are evaluated.

[3]https://jira.mariadb.org
[4]https://bugs.mysql.com/search-csv.php

B.4 LLM Classification

After performing the keyword-based filtering and analysis, we realized that the produced set contained a large amount of misclassified bugs/CVEs. Therefore, for each bug or CVE we construct a structured prompt and submit it to a locally hosted *gpt-oss-120b* model[5] (Q4_K_M quantization) served by *llama-server* [49] on an H100 GPU with continuous batching enabled and 16 parallel completion slots. Descriptions are truncated to 12,000 characters to stay within the model's context budget. Up to 16 classification requests are dispatched concurrently per run. The system and user prompts for bug reports and CVEs are shown below.

Bug Report Classification Prompt

System: You are an expert software security researcher specializing in database systems.
User: Analyze the following bug report.
First, classify the issue into one of the following types: 1. "*bug*": Actual bugs (flaws in the system implementation). 2. "*programmer_error*": Problems on the side of the programmer (misuse). 3. "*other*": Requests or other issues (feature requests, questions, etc.).
Second, IF the issue is a "*bug*", determine if it describes a **Memory Safety Issue**. If it is not a "*bug*", "*is_memory_safety*" must be false.
Third, IF the issue is a "bug" AND you can determine the primary operational implication with HIGH confidence, classify it into exactly one of the following:

- *"data_loss"*: Data is permanently lost or unrecoverable.
- *"data_corruption"*: Stored data becomes physically or logically inconsistent.
- *"incorrect_query_results"*: Queries silently return wrong data (no crash).
- *"system_crash"*: The database process crashes, aborts, or panics.
- *"resource_exhaustion"*: Memory leak, disk/CPU saturation, connection pool depletion.
- *"concurrency_hazard"*: Deadlocks, race conditions, or isolation violations under concurrency.
- *"transaction_violation"*: ACID guarantees are broken (lost updates, dirty reads, etc.).
- *"durability_violation"*: Committed transactions are lost after a crash or restart.
- *"replication_inconsistency"*: Primary and replica data diverge or replication fails.
- *"denial_of_service"*: Server becomes unresponsive or hangs without crashing.
- *"none"*: No clear operational implication can be determined with high confidence, or the issue is not a "bug".

Bug Report:
Title: [title_placeholder]
Description: [description_placeholder]
Task:
Classify this bug. Answer with JSON format:

```
{"issue_type": "...",
 "is_memory_safety": true/false,
 "implication": "...",
 "confidence": "...",
 "reason": "..."}
```

Only provide the JSON."

[5]https://huggingface.co/unsloth/gpt-oss-120b-GGUF

CVE Classification Prompt

System: You are an expert software security researcher specializing in database systems and CVE analysis.
User: Analyze the following CVE (Common Vulnerabilities and Exposures) report.
First, classify the issue into one of the following types: 1. "*bug*": Actual bugs (flaws in the system implementation). 2. "*programmer_error*": Problems on the side of the programmer (misuse). 3. "*other*": Requests or other issues (feature requests, questions, etc.).
Second, IF the issue is a "*bug*", determine if it describes a **Memory Safety Issue**. If it is not a "*bug*", "*is_memory_safety*" must be false. Memory safety issues INCLUDE:

- Buffer overflow, heap overflow, stack overflow
- Use-after-free, double-free, dangling pointer
- Null pointer dereference
- Out-of-bounds read/write
- Uninitialized memory access
- Integer overflow ONLY if it leads to memory corruption

Memory safety issues DO NOT INCLUDE:

- SQL injection, XSS, CSRF (these are injection/logic bugs)
- Authentication bypass, privilege escalation (unless memory-related)
- Denial of service through resource exhaustion (unless memory corruption)
- Logic errors, race conditions (unless causing memory corruption)

Third, IF the issue is a "bug" AND you can determine the primary operational implication, with HIGH confidence, classify it into exactly one of the following:

- *"data_loss"*: Data is permanently lost or unrecoverable.
- *"data_corruption"*: Stored data becomes physically or logically inconsistent.
- *"incorrect_query_results"*: Queries silently return wrong data (no crash).
- *"system_crash"*: The database process crashes, aborts, or panics.
- *"resource_exhaustion"*: Memory leak, disk/CPU saturation, connection pool depletion.
- *"concurrency_hazard"*: Deadlocks, race conditions, or isolation violations under concurrency.
- *"transaction_violation"*: ACID guarantees are broken (lost updates, dirty reads, etc.).
- *"durability_violation"*: Committed transactions are lost after a crash or restart.
- *"replication_inconsistency"*: Primary and replica data diverge or replication fails.
- *"denial_of_service"*: Server becomes unresponsive or hangs without crashing.
- *"none"*: No clear operational implication can be determined with high confidence, or the issue is not a "bug".

CVE Report:
CVE ID: [CVE_ID placeholder]
CWE: [CWE_IDs placeholder]
Affected: [Vendor/Product placeholder]
Description: [Description placeholder]
TASK:
Classify this CVE. Answer with JSON format:

```
{"issue_type": "...",
 "is_memory_safety": true/false,
 "implication": "...",
 "confidence": "...",
 "reason": "..."}
```

Only provide the JSON."

The model produces a JSON object with five fields:

(1) **issue_type**: one of *bug*, *programmer_error*, or *other*.
(2) **is_memory_safety**: a boolean, forced to *false* if the *issue_type* is not a *bug*.
(3) **implication**: the primary operational consequence, constrained to one of: *data_loss*, *data_corruption*, *system_crash*, *incorrect_query_results*, *resource_exhaustion*, *concurrency_hazard*, *transaction_violation*, *replication_inconsistency*, *durability_violation*, *denial_of_service*; *none* if no implication can be determined with high confidence or the entry is not classified as a *bug*.
(4) **confidence**: one of *high*, *medium*, or *low*.
(5) **reason**: a brief free-text explanation of the classification.

To reduce sensitivity to non-deterministic model outputs, each classification run is repeated *three times independently*, producing three result sets. In our analysis, a bug is counted as a *memory safety bug* only when **all** three repetitions agree (i.e., the intersection of the three LLM sets).

B.5 Result Analysis

Our analysis scripts produce a set of complementary statistics that we collect to gain confidence in the results presented in §2.1. The full outputs are retained for validation and manual inspection. More precisely, our analysis produces two types of output:
Aggregate statistics. For each database and each repetition, we extract: the total number of entries processed; the count classified as *bug*, *programmer error*, or *other*; and the subset flagged as a memory safety bug. The keyword set size is gathered in parallel as an independent baseline. Tracking these numbers across all three repetitions allows us to assess the stability of the classifier and identify any significant variation between runs.
Intersection analysis & result validation. We compute all non-empty intersections across the four signal sets (Keywords, LLM repetition 1, LLM repetition 2, LLM repetition 3), to observe the *exclusive* contribution of each subset combination (i.e., entries that appear in exactly that combination and no other). This reveals the degree of agreement between the keyword heuristic and the LLM classifier, and quantifies how consistently each entry is flagged across repeated classifications. Note that the keyword set serves here as a comparative signal only; the **confirmed** memory safety count is derived exclusively from the three-way LLM intersection, as this produced more accurate results. Finally, to further validate the automated classification, we conducted a manual inspection of a sample of flagged entries, cross-checking titles, descriptions, and LLM outputs against the expected classification.
Limitations. This pipeline is a motivating analysis, not a validated vulnerability classifier, and its numbers should be read as estimates. We use a quantized (Q4_K_M) *gpt-oss-120b* model rather than a full-precision frontier model, and we do not report precision, recall, or inter-annotator agreement against a manually labeled ground-truth set; dedicated works such as DUALLM [81] provide such validated baselines. Our three-way agreement rule—counting a report as a memory-safety bug only when all three runs concur—has two distinct effects. The first concerns *coverage*: requiring agreement filters out reports that only one or two runs flagged, i.e., the cases the model classifies inconsistently. These borderline cases are dropped

from the counted set, so the rule trades away some coverage in exchange for a set of entries that is more likely to be correct. The second concerns *confidence*: one might hope that three runs act like three independent judgments, so that a mistake in one run is caught by the others. This does not hold here, because we sample the same model three times. When such a model is uncertain about a case, it tends to make the *same* error on every run rather than a different one each time, so the runs are correlated rather than independent [28]. Agreement across runs therefore reflects the model's consistency, and provides less assurance than three genuinely independent judgments would. We also classify issue-tracker reports and CVEs only, and exclude fixes that land as pull requests without an associated issue (Appendix B.2); issue-text-only classification is known to be weaker than classification that also ingests commit messages and diffs [81]. Consequently, our claims concern memory-safety *reports* rather than all memory-safety bugs. Finally, MySQL's bug tracker redacts security-sensitive issues, so memory-safety bugs for MySQL are systematically under-reported in the source we scrape, which partly explains its low memory-safety share in §2.1.

Received 17 April 2026; revised 19 August 2026; accepted 12 September 2026