PTSan: A Practical Memory Safety Sanitizer for C/C++ with Pointer-Object Authority

Eli Davis, Eric Lahtinen, and Michael Gordon
Aarno Labs
Boston, MA, USA
{eli, elahtinen, mgordon}@aarno-labs.com

***Abstract*—Memory safety errors remain the dominant source of severe vulnerabilities in C and C++. Pointer-based sanitizers provide stronger guarantees than location-based tools such as LLVM's ASan, but their overhead and compatibility limitations have constrained production use. We present PTSan, an LLVM sanitizer that makes pointer-based checking practical by storing an object identifier in each pointer's high bits and its bounds in a fixed-size runtime table. This representation trades a finite live-object budget for low overhead, optimizer visibility, and commodity-hardware support. Because identity travels with the pointer value, ordinary LLVM dataflow propagates it without explicit per-pointer metadata instructions. Separating metadata lookup from check enforcement exposes both as LLVM IR, enabling check hoisting, elision, and merging, plus whole-function min-cut placement of compatibility tag strips. Intel Linear Address Masking eliminates the remaining strips in hardware when available.**

On stock hardware PTSan runs at parity with the fastest published location-based sanitizer: 57.2% geomean overhead on SPEC CPU 2017 on x86-64 (46.4% with Intel LAM), 54.7% on ARM64, and 31.5% (x86-64) on the application-shaped LLVM MultiSource suite. This is roughly a third of the published overhead percentage of prior systems with similar pointer-object authority guarantees, while preserving the inter-object, non-object, and temporal detection coverage measured by an independent memory safety test suite. Its physical-memory overhead is effectively native, a critical property for production deployment as memory costs become a first-order constraint. We also demonstrate practical overhead on real-world server and security workloads. These results show that PTSan brings practical pointer-based memory safety into a recompile-only sanitizer deployment model.

1. Introduction

Memory safety remains the central security problem of deployed systems software. C and C++ still implement operating systems, browsers, language runtimes, databases, and performance-sensitive servers, and the manual bounds and lifetime discipline they require routinely fails: memory-safety errors account for the majority of severe vulnerabilities reported by major vendors [1], including about 70% of Chromium's high-severity bugs [2], and they are the substrate for code-reuse attacks, data-only attacks, information disclosure, and remote compromise [3, 4]. The problem is accelerating: LLM agents already make exploiting known vulnerabilities faster and cheaper [5], and they are beginning to find and exploit previously unknown, unpatched ones [6]. Because these codebases cannot be rewritten in memory-safe languages quickly, they need protection that arrives before each individual bug is found, triaged, and patched.

Sanitizers are the natural vehicle for that protection, adding memory-safety guarantees by recompilation, without source changes. Redzone sanitizers, the most popular category, are ABI compatible, and the model is proven: AddressSanitizer (ASan) is used throughout industry to catch memory errors [7]. But their own authors position them as testing tools, not deployed defenses, and not by accident: they attach metadata to memory locations and pay for that compatibility twice. First in guarantee: ASan and the faster RangeSanitizer (RSan) [8] check whether an address is valid, not whether the pointer is entitled to access it, so an access redirected into another live object passes their checks, exactly the freedom an adaptive attacker needs. Second in memory: redzones, shadow memory, and quarantine triple physical memory use (3.33× for ASan and 3.01× for RSan on SPEC CPU 2017), disqualifying them in production fleets where memory is a first-order cost. Pointer-based systems close the guarantee gap by binding each pointer to the object it may access, but at costs that have kept them out of deployment: SoftBound+CETS breaks the ABI by passing pointer bounds as extra function arguments, and runs at 161% overhead with 2.7× memory [9]; CUP costs 158% [10].

We present PTSan, an LLVM sanitizer that provides pointer-based checking at redzone-sanitizer speed and near-native memory. PTSan stores a compact object identifier in the high bits of each pointer; the identifier indexes a flat runtime table holding the object's base and span. Before a memory access, PTSan checks that the entire access range lies within the bounds of the object the pointer names. When an object is freed, its entry is cleared. Once identifiers are reused, a stale pointer is accepted only if a replacement object at its address receives the same identifier, providing probabilistic temporal protection.

Two properties make PTSan fast where prior pointer-based systems were slow. Identity propagation is free: the

identifier lives in the pointer value, so ordinary dataflow (copies, arithmetic, pointer loads and stores) carries it without metadata instructions. And the work that remains at an access stays ordinary LLVM IR, with bounds loads kept separable from the checks they feed, so the compiler can hoist identifier extraction and bounds loads, merge and elide checks, and replace per-iteration loop checks with whole-loop range checks computed before LLVM's loop optimizations restructure the code.

PTSan's remaining design decision is a deliberate compatibility tradeoff: it supports 2^{16} simultaneously live object identifiers. The default width follows from ordinary x86-64 allocation: `glibc`'s `malloc` returns addresses in the low 48-bit user address space, leaving 16 pointer bits in which PTSan can carry object identity. PTSan can therefore retain the system allocator unchanged rather than requiring a custom allocator or partitioned address space. The compact identifier also bounds metadata to roughly a fixed table of 1.2 MiB, with no redzones, shadow memory, or quarantine. Support for Intel's Linear Address Masking (LAM) [11] is a further benefit of the design: LAM_U48 masks 15 identifier bits during memory accesses, removing most software tag stripping while reducing the ID budget to 2^{15}. The cost is a finite live-object limit, but the budget covers most programs we measured: 91.0% of the 167 LLVM MultiSource programs fit the default configuration. Deployments can measure peak demand in advance, and strict mode fails closed if the budget is exceeded.

Our overhead results[1] move pointer-based memory safety into a new performance regime. On SPEC CPU 2017, PTSan's geomean runtime overhead is 57.2% on stock x86-64 hardware: parity with RSan, the fastest published redzone sanitizer, roughly half of ASan's overhead on the same machine, and a third of the published overhead percentage of prior pointer-based systems. When Intel LAM elides the tag stripping that remains after the compiler has hoisted most of it, the geomean falls to 46.4%. The result is portable: recompiling for ARM64 yields 54.7% on the same suite, with no hardware assist. SPEC's pointer-dense compute kernels are a stress test rather than PTSan's target workload; on the broader 149-program LLVM MultiSource suite the geomean is 31.5% on x86-64 (27.2% with LAM, 40.8% on ARM64), and seven real-world server and cryptographic workloads (Redis, PostgreSQL, NGINX, OpenSSL, memcached, Apache, SQLite) add overheads ranging from near-native to 44% on x86-64, each well within the 2^{16} identifier budget (peak demand 16,411).

PTSan's physical-memory overhead is largely fixed and measured at 1.014× geomean with a worst case of 1.121× on SPEC. PTSan is effectively native, where ASan costs 3.33× and RSan 3.01× on the same benchmarks and baseline, and SoftBound+CETS self-reports the cost at 2.7× [9]. For production deployment this is the decisive number: PTSan's footprint is bounded by design and measured near-native, at the cost of only a slice of applicability.

PTSan's strong safety guarantee is also measurable. On MSET [12], an independent benchmark suite for assessing memory-safety sanitizers, PTSan matches the originally published coverage of SoftBound+CETS, the strongest prior pointer-based system, detecting all 126 inter-object spatial and all 40 temporal cases; it is the only low-overhead system that detects non-linear out-of-bounds accesses that land inside another live object, an access pattern available in real-world vulnerabilities.

This paper makes the following contributions:

Design. A sanitizer design that carries a compact object identifier in the pointer and keeps bounds in a small fixed table. This design makes identity propagation free, checks ordinary optimizable LLVM IR, and memory overhead bounded by construction. We present a correctness argument covering both the base instrumentation and its optimizations.

Optimizations. Compiler analyses that exploit the separability of bounds loads from checks: identifier-extraction and bounds-load hoisting, range-check merging and elision, loop preanalysis whose results survive LLVM's loop transformations, and a min-cut placement of tag strips and retags that hoists stripping out of hot loops.

Implementation. A complete LLVM/compiler-rt sanitizer for x86-64 and AArch64 covering stack, heap, global, and mmap allocation, libc interposition, and software or LAM-accelerated identifier stripping, released as open source at `https://github.com/Aarno-Labs/PointerTagSanitizer`.

Evaluation. Across SPEC CPU 2017, 149 LLVM MultiSource programs, MSET, and real server workloads, on two architectures: runtime overhead at parity with the fastest redzone sanitizer, near native memory overhead, inter-object and temporal detection coverage matching the strongest pointer-based system, and identifier demand that validates the bounded-ID design for its target class of programs.

2. Background

Memory-Safety Errors and Checking Models. Memory-safety violations are usually divided into spatial and temporal errors. A *spatial* error occurs when a program accesses memory outside the object that a pointer is allowed to reference. Following MSET, we further distinguish where the illegal access lands: an *inter-object* access lands inside a different currently allocated object, a *non-object* access lands in unallocated or padding memory, and an *intra-object* access stays within its allocation but crosses an internal field or subobject boundary [12]. A *temporal* error occurs when a pointer is used after the lifetime of its object has ended, or when a deallocation operation does not match the object being deallocated. In this paper, unless stated otherwise, *object* means an allocation object: a heap allocation, stack allocation, global object, or mmap allocation. PTSan enforces bounds at this granularity and does not claim intra-object field protection (Section 5.5).

1. We report full-suite overheads including ID-exhausting workloads. These runs fall outside PTSan's strong-guarantee regime, but ID propagation and access checking remain active. We retain them so the aggregates capture instrumentation cost across the full evaluated suite.

Sanitizers for these errors follow one of two checking models. A *location-based* sanitizer attaches validity metadata to memory addresses and asks, at each access, "is this target address valid?"; ASan's poisoned redzones are the canonical example [7]. A *pointer-based* sanitizer associates each pointer with the object from which it was derived and asks, "is this pointer allowed to access this range?"; SoftBound+CETS is the canonical example [13, 14]. The models differ exactly when an illegal access lands in valid memory: a location-based check accepts an access that escapes its object and lands inside another live object, while a pointer-based check rejects it, because the authority carried by the pointer still names the original object. PTSan is a pointer-based sanitizer, and such inter-object errors arise in real vulnerabilities, as the next subsection illustrates.

A Recent Inter-Object Overflow. CVE-2025-57807, a critical (CVSS 9.8) heap overflow in ImageMagick 7.1.2-0, shows why this distinction is security-relevant in current C code [15]. A `BlobStream` forward seek advances the write offset past the buffer's capacity; the next write grows the buffer by the write length rather than by the full distance to the offset, then forms its destination by adding the attacker-controlled offset to the buffer base. The write therefore lands wherever the offset points, not merely past the end of the source buffer. Its ingredients are mundane: an offset or index an attacker can influence, and a resize that fails to account for it, a pattern that recurs throughout buffer, stream, and serialization code. When the offset is chosen to land inside a different live allocation, the destination is valid, mapped memory, and address- or range-validity checks have nothing to reject.

Building a working exploit was straightforward. A small harness drives the vulnerable `WriteBlob` path directly and steers the write into a second live heap object; reproducing it under ASan took only modest tuning, an oversized backing allocation and a predicted `realloc` relocation offset to absorb ASan's allocator perturbation. Both ASan and RSan [8] then accept the corrupting write, since its destination is an allocated, address-valid object, while PTSan detects it because the writing pointer's object ID authorizes the source buffer, not the object the write lands in.

3. Threat Model

This section states the assumptions under which PTSan makes its security claims and the attacker it defends against. Once the mechanism is defined (Section 4), Section 5 gives the security analysis.

3.1. Assumptions

We assume the toolchain is correct and the instrumentation is complete: the compiler, the C library, and PTSan itself implement their specifications, and the relevant allocations, deallocations, and memory accesses are instrumented or intercepted. Completeness is the practical limit of a young sanitizer rather than a property of the design; uninstrumented allocation sites and unsupported operations are discussed in Section 5.5.

The remaining assumptions are structural. First, the program has at most 2^{16} live objects at any one time, or 2^{15} when Linear Address Masking is enabled, so every live object can hold a distinct identifier; in strict mode, identifier exhaustion fails closed. Second, program addresses fit in the low 48 bits, leaving the high 16 bits for the identifier (Section 4.1). Third, address-space layout randomization (ASLR) is enabled. For temporal claims we additionally assume the program is free of data races between freeing an object in one thread and accessing it in another.

We distinguish strict and permissive modes. Strict mode is the configuration for the security claims in this paper: identifier exhaustion and accesses through the untagged (zero) or default identifier fail closed. Permissive mode keeps such accesses running and is an engineering aid for bring-up and compatibility debugging, not a security configuration.

3.2. Attacker Model

We assume an attacker who can trigger memory-safety bugs in instrumented C/C++ code by controlling program inputs: offsets, lengths, indices, object contents, allocation sizes, and control flow along vulnerable paths. Two independent axes describe such an attacker: how much of the system the attacker knows and controls, and what bug primitive the vulnerability gives them. The security analysis (Section 5) walks this grid.

3.2.1. Knowledge and environment control. We distinguish three attacker levels, from strongest to weakest:

- **A1 (omniscient, controls all allocation).** The attacker understands PTSan and the program perfectly and can predict exactly which identifier each allocation receives. This is the worst case for PTSan, and it is also the least realistic: it requires the attacker to be able to model all allocations throughout the program's life, which is difficult for a program serving other requests.
- **A2 (omniscient, cannot control allocation).** The attacker understands PTSan and the program but there is enough noise in the system that they cannot predict identifier assignment. This is the realistic strong attacker.
- **A3 (understands the program, not PTSan).** The attacker understands the target program but is not aware of the specific defense.

3.2.2. Bug primitives. Orthogonally, we distinguish what the vulnerability lets any of these attackers do:

- **P1 (length control).** The attacker controls the length of a contiguous access through a legitimately derived pointer, such as a length passed to `memcpy` or the trip count of a loop that advances element by element. The accessed range grows outward from the source object and cannot skip over adjacent memory.
- **P2 (offset control).** The attacker controls an index, offset, or stride combined with a legitimately derived pointer, so

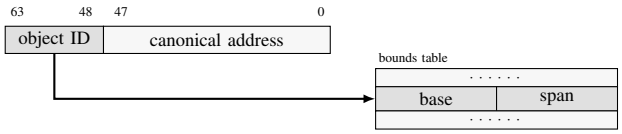


Figure 1. PTSan pointer layout: the 16-bit object ID indexes a flat table of the object's base and span; the low 48 bits remain the canonical address.

the resulting access can land at a discontiguous address, including inside another live object. This is the shape of the non-linear out-of-bounds errors in MSET's taxonomy and of the ImageMagick vulnerability of Section 2.

- **P3 (pointer forgery).** The attacker constructs full pointer values, including the high bits, through integer-to-pointer manipulation or arithmetic with very large numbers. A fabricated value did not obtain its identifier from an allocation, so the dataflow argument for checked accesses no longer applies, and we analyze this primitive separately (Section 5.3).
- **P4 (temporal primitives).** The attacker holds a stale pointer to a freed object and can influence when the program allocates and frees objects.

PTSan's primary security property is object-bounds enforcement for checked accesses: a memory access through a tagged pointer must lie within the live allocation object named by the pointer's identifier. The next sections define the mechanism and then analyze what each attacker level achieves with each primitive.

4. PTSan Design

PTSan is a recompile-only LLVM sanitizer for existing 64-bit C/C++ programs whose addresses fit in the low 48 bits: protected code is compiled with PTSan instrumentation, linked against the PTSan runtime, and run unchanged. It is built around a simple representation: each protected pointer carries the identity of the allocation object from which it was derived. The runtime stores bounds and lifetime state for that object in compact tables indexed by the ID. A memory access therefore checks the object named by the pointer, rather than asking only whether the target address is currently mapped or surrounded by poisoned memory.

4.1. Tagged Pointer Representation

PTSan uses the upper 16 bits of a 64-bit pointer as an object ID and the lower 48 bits as the canonical program address (Figure 1). The implementation uses two reserved identifiers. ID 0 names untagged pointers. The highest ID is a default ID used by permissive mode when the runtime cannot assign a normal object ID. The remaining IDs are assigned to protected stack, heap, mmap, and global objects. In strict mode, untagged and default-ID pointers are invalid, allowing for the strong security guarantee; in permissive mode, they are open as a compatibility fallback.

Embedding the ID in the pointer is the main reason PTSan avoids a separate metadata-propagation problem. LLVM copies, PHI nodes, select instructions, bitcasts, loads and stores of pointer values, and derived pointer computations move the full pointer value. Because the ID is part of that value, ordinary dataflow carries it along with the address. A pointer derived from an allocation therefore continues to name the same allocation even when its low address changes through pointer arithmetic, and the work that remains at a dereference (identifier extraction, bounds loads, and range checks) stays ordinary LLVM IR that the compiler can optimize (Section 6).

When a raw address is required, PTSan removes the high bits before the address is observed. The runtime provides an ID-stripping pass and call-boundary handling for operations that must receive canonical pointers. By default this strip is a software mask inserted by the compiler and runtime; Section 6.7 describes how the compiler places these strips to minimize how often they execute. On systems with Intel Linear Address Masking (LAM) [11], the hardware ignores the tag bits during address translation, so PTSan can carry the identifier through most memory operations without an explicit per-access strip. The compiler and runtime still strip at boundaries where software expects canonical addresses, such as pointers passed to the kernel. PTSan uses the LAM_U48 variant, which masks bits 62–48, so the LAM configuration uses a 15-bit identifier: bit 63 is reserved by the hardware mask, and the live-object budget is correspondingly smaller. Because ordinary x86-64 program addresses occupy only the low 48 bits [16], masking these bits leaves the canonical address intact. On AArch64, PTSan uses the full 16-bit identifier and strips in software throughout: the architecture's Top-Byte-Ignore feature masks only the top eight bits, too few for PTSan's identifier, so PTSan does not use it. We report all three configurations (x86-64, x86-64 with LAM, and ARM64) in Section 8.

4.2. Runtime Metadata

PTSan stores object bounds in a fixed-size runtime table. Each ID indexes a bounds entry containing two 64-bit fields: the first has the ID in its upper 16 bits and the base address in its lower 48 (`base`), and the second is the size of the allocation (`span`). Metadata retrieval is therefore a direct index into a flat table, with no shadow-memory walk or multi-level lookup. The runtime also maintains ID-pool state for non-stack objects and thread-local state for stack IDs.

The bounds entry is the authoritative metadata for an object. When an object is live, its entry contains the above struct. When the object is no longer live, PTSan clears the entry, setting base and span to zero. This single transition is used by both spatial and temporal checks: spatial checks compare an access range against the live bounds, while immediate stale dereferences fail because the span is zero.

4.3. Allocation Instrumentation

PTSan assigns IDs at the points where allocation objects enter the protected program. For heap and mmap objects, sanitizer interceptors wrap allocation APIs such as `malloc`, `calloc`, `realloc`, and `mmap`. On allocation, the wrapper

obtains an ID, writes the object's base and span into the metadata table, and returns the pointer with the ID in the upper bits. If an operation changes an allocation's size, the runtime updates the span for the object's ID. By default, the recorded span is the requested allocation size rather than the allocator's usable size, so PTSan also rejects accesses that stray into allocator slack beyond the request.

For stack objects, the LLVM pass instruments eligible `alloca` instructions. At function entry, PTSan obtains the IDs needed for the function's stack allocations and writes their bounds. The common case uses thread-local contiguous ranges of stack IDs, avoiding a runtime call for each function that contains at least one `alloca`. If the fast path cannot supply the requested IDs, the pass calls a runtime helper that allocates and initializes the IDs. At function returns and other exits, including unwind paths, the instrumentation returns the IDs to the runtime. In PTSan's stack-temporal mode, it also clears the corresponding bounds entries and batches pending stack-ID returns so that consecutive calls to the same function do not immediately reuse the same stack IDs; delaying reuse this way makes stack-buffer-leak detection more robust.

Globals are handled by introducing tagged global-pointer proxies. At startup, a PTSan constructor assigns IDs to instrumented globals, writes their bounds, creates tagged pointer values, and stores those values in the proxies. Uses of the original global in instrumented code are redirected through the proxy, so derived pointers carry the global's ID in the same way as heap and stack pointers. External or unsupported globals can be represented conservatively, but the object-bounds guarantee of Section 5 applies to globals for which PTSan creates precise IDs and bounds.

4.4. Access Checks

PTSan instruments LLVM memory operations with range checks. The pass identifies pointer operands of scalar loads and stores, contiguous vector loads and stores, supported atomic operations, compare-and-exchange operations, and memory intrinsics such as `memcpy`, `memmove`, and `memset`. For fixed-width operations, the access size comes from the LLVM type. For memory intrinsics, the size operand defines the checked range, so a variable-length copy or set requires one range check per pointer operand rather than one check per byte.

For a pointer `ptr` and access size `size`, the base check is:

```
id     = ptr >> 48;
base   = ptsan_bounds[id].base;
span   = ptsan_bounds[id].span;

ok = (size <= span) && (ptr - base <= span - size);
```

This check validates an access range, not only the first accessed address. The first comparison ensures that the requested access can fit within the object at all. The second comparison verifies that the low address is no earlier than the object's base and that the range ending at `ptr + size` does not exceed the object bound. As a result, PTSan detects both ordinary out-of-bounds offsets and type-width errors in which a pointer to a small object is used for a larger access.

On failure, the inserted code branches to a reporting path that calls the PTSan runtime with the check site, pointer ID, access type, address, and size. The passing path is straight-line integer arithmetic, metadata loads, and conditional branches. This is the mechanism that later optimizations operate on: the pass can eliminate or move checks, but the base check defines the condition that an access must satisfy.

4.5. Deallocation and Temporal Enforcement

Deallocation is the inverse of allocation. A free-like operation extracts the ID from the pointer and checks that the ID names a live object whose recorded base address matches the pointer. If it does not, PTSan reports an invalid or double free. If it does, PTSan clears the bounds entry and returns the ID to the appropriate pool.

Clearing bounds means a stale nonzero-size dereference fails deterministically until the ID is reused. A stale pointer still carries its old ID, but that ID's span is zero, so the range check fails. After ID reuse, temporal protection is probabilistic. A stale pointer can pass only if the reused ID now describes a live object whose base address matches it. This still requires the stale pointer to line up with both the reused ID and the new object's address, but it is not a deterministic temporal safety guarantee. Section 5.4 analyzes how hard that coincidence is to arrange: for an attacker who cannot predict identifier assignment lining up reuse and overlap succeeds only with very low probability.

4.6. Interoperability Boundaries

PTSan is designed for deployment on existing C/C++ codebases, so it has to coexist with code that expects ordinary pointers. The compiler pass strips IDs before operations and calls that require canonical addresses. The runtime also tracks when execution is inside libc so that internal allocator and libc activity does not accidentally create application-visible tags.

The important distinction is between security boundaries and compatibility boundaries. Allocation and deallocation wrappers are security-critical because they create, resize, and retire object metadata. Generic libc wrappers and ID stripping are compatibility mechanisms because they prevent high-bit tags from leaking into code that cannot interpret them. If uninstrumented code performs a memory access after receiving a stripped pointer, that access is not covered by PTSan's core check (Section 5.5).

5. Security Analysis

The base guarantee is for checked LLVM IR memory operations. Assuming that allocation and deallocation events are instrumented or intercepted, that every in-scope memory operation is instrumented, that strict ID mode is used, and that high pointer bits are not forged by explicit integer-pointer manipulation, every checked nonzero-size access

TABLE 1. OUTCOMES PER ATTACKER LEVEL AND BUG PRIMITIVE.

Primitive	A1	A2	A3
P1: length	blocked	blocked	blocked
P2: offset	blocked	blocked	blocked
P3: forgery, heap	blocked under ASLR	blocked under ASLR	blocked under ASLR
P3: forgery, stack	stack variables only†	stack variables only†	blocked
P4: temporal primitives	reuse grooming	$\sim 2^{-16}$/attempt	$\sim 2^{-16}$/attempt

†Confined to user-declared variables; saved return addresses and other system data are unreachable (Section 5.3).

through a PTSan pointer either lies within the live allocation object named by that pointer's ID or traps before the access executes. Table 1 summarizes the outcomes.

The claim is object-granular. PTSan tracks heap allocations, stack allocations, globals, and mmap regions as allocation objects. It does not claim subobject safety. The claim also applies only at instrumented or intercepted boundaries: stripped-pointer accesses in fully uninstrumented code are outside the core guarantee.

5.1. Runtime Invariants

The base argument relies on four invariants that map directly to PTSan's runtime metadata and instrumentation.

Allocation metadata. For each live tracked object, the runtime records a nonzero object ID, a low-address base, and a span. The bounds entry for that ID describes exactly the allocation object for which the ID was assigned.

Live-ID uniqueness. A normal ID names at most one live object at a time. In strict mode, ID 0 and the default ID do not authorize ordinary accesses. When no normal ID is available, the allocation receives the default ID, which carries no bounds, so any access through it traps; strict mode therefore fails closed rather than granting broad bounds.

Lifetime invalidation. When a tracked object's lifetime ends, PTSan clears the object's bounds entry before the ID can authorize later accesses. Heap and mmap deallocation paths also check that the pointer being deallocated names the recorded base address before clearing metadata.

Dereference mediation. In the unoptimized transformation, every protected LLVM memory operation is preceded by a check over the same pointer authority and an access range that covers the operation. This includes scalar loads and stores, supported vector and atomic operations, LLVM memory intrinsics such as `memcpy`, `memmove`, and `memset`, and supported libc memory accesses such as `strdup` and `wmemcpy`. Unsupported intrinsics, inline assembly, and memory effects hidden inside unsupported uninstrumented calls are coverage boundaries (Section 5.5). All optimizations preserve the security of the original unoptimized transformation.

5.2. Spatial Safety

The spatial guarantee follows by composing the invariants. PTSan stores object authority in the pointer value itself: an allocation source writes the object's ID into the high bits, and LLVM pointer-preserving operations (bitcasts, PHI nodes, selects, loads and stores of pointer values, and pointer arithmetic that does not overflow into the ID bits) carry that ID along with the low address, with no separate shadow state. So if a pointer value is derived from object `obj` by in-scope pointer operations, the ID extracted from it remains `id(obj)`. Allocation instrumentation establishes the metadata and live-ID uniqueness invariants, and dereference mediation ensures the memory operation cannot execute until the check (the interval-containment check of Section 4.4) has loaded the metadata for that ID and proved that the full access range lies inside the recorded bounds. Therefore, under the scope assumptions, a checked access through a protected tagged pointer cannot reach outside the allocation object from which the pointer was derived.

For the P1 and P2 primitives, this guarantee is attacker-independent. The identifier is fixed by the source allocation and travels with the pointer, so no choice of length, index, or offset escapes the source object. A3 gains nothing from understanding the program, A2 gains nothing from also understanding PTSan, and A1's control over allocation does not help either: spatial enforcement never depends on which identifier an object receives, only on the binding between the pointer and its object's recorded bounds. Against all three levels, P1 and P2 reduce to accesses within the source object. The one escape route is a P2 offset large enough to carry past bit 47 into the identifier field; that carry changes the identifier, which makes the access a forgery attempt, covered by the case analysis of the next subsection.

The P1/P2 split is where location-based checking ends: redzones stop a P1 sweep at the object boundary, but a P2 access that jumps into another live object passes an address-validity check, and corrupting non-control data reachable this way suffices for arbitrary computation [17]. PTSan does not distinguish the two primitives: both reduce to the same interval-containment check against the source object's bounds.

5.3. Pointer Forgery and Address-Space

The P3 primitive removes the dataflow assumption: if the attacker can write the high bits directly, a pointer's identifier no longer necessarily came from its source allocation, but the runtime check is still executed. A forged high-bit identifier access of `addr` has four possible outcomes. An unused identifier fails because its bounds are empty; a live identifier fails if `addr` is outside of the object it names; the zero and default identifiers fail in strict mode; and the only passing case is when the identifier belongs to an object that contains `addr`. In that last case the attacker has reconstructed a valid pointer to that object, not an arbitrary-address write primitive.

A passing forgery therefore requires a correct identifier–address pair, and the attacker levels differ in how they can obtain one. A3 does not know this is a possibility and thus is blocked. A2 knows PTSan's layout but cannot predict identifier assignment, and under ASLR it does not know where any given object lives; recovering either piece of information requires an information leak, which on these programs typically means an out-of-bounds read, exactly

the access class PTSan already blocks. A1 can do better: by controlling all allocation it can predict which identifier names which object, so it can supply a valid identifier. But it still needs the object's address, so for heap objects A1 stands where A2 stands: blocked until ASLR is defeated. This is a stronger position than ASLR alone provides. Under ASLR without pointer authority, an attacker who can write arbitrary addresses can still groom the heap and corrupt whatever lands at a chosen address; under PTSan, a write succeeds only if the attacker already knows the specific object and address it targets.

Stack objects are the exception. ASLR randomizes the stack base but not the relative layout of stack slots, so attackers A1 and A2, who can forge a stack identifier and compute addresses relative to a stack pointer they hold, may reach another stack variable in the same region. PTSan still confines such an attacker to user-declared stack variables, which excludes saved return addresses, so it does not yield direct control-flow hijacking. Combining PTSan with stack-layout randomization [18, 19] would recover for stack objects the property ASLR gives heap objects.

5.4. Temporal Safety

The P4 primitive exercises the same check but depends on identifier reuse, and PTSan's temporal protection is probabilistic even under the full assumption list. When an object is freed, PTSan checks that the freed pointer names the object's recorded base address, clears the bounds entry, and returns the identifier to the runtime pool. A stale pointer keeps its old identifier, but the cleared bounds give it a zero span, so any nonzero-size dereference fails until the identifier is reused. A double or invalid free fails for the same reason: the second deallocation no longer finds a live object with a matching base for that identifier.

The residual risk is identifier reuse, and the attacker levels differ exactly in their ability to arrange it. Identifier assignment is deterministic: the non-stack ID pool is a FIFO ring, so a freed identifier is not reassigned until the ring cycles, on the order of 2^{16} intervening allocations. Reuse is therefore never immediate, and short-lived stale pointers, the common case in practice, are caught with certainty. The same determinism is the worst-case weakness: A1, which can model every allocation and free, knows which identifier each allocation receives and can time a victim allocation to receive the stale identifier at an overlapping address. Learning the controlled pointer's ID requires either being the only source of allocation activity for the program's lifetime or an information leak, which PTSan blocks; this is what makes A1 unrealistic. A2 and A3 cannot observe the ring's position, so to them the identifier carried by any object overlapping their stale address is effectively uniform over the identifier space: hitting the stale identifier requires on the order of 2^{15} to 2^{16} allocation-and-probe events, and every failed probe is a detected violation.

We do not count address overlap as a further line of defense: re-tenanting the stale address is the premise of exploiting a use-after-free, and allocators make it routine. PTSan's barrier is orthogonal to placement: the re-tenanted victim carries whatever identifier the pool dispenses while the stale pointer keeps its old one, so the identifier analysis above is the whole story.

The practical guarantee is therefore high-probability detection of use-after-free, double-free, and invalid-free errors against A2 and A3, defeated only by the highly unrealistic A1. Randomized identifier reuse would make the assignment schedule unpredictable even to an attacker who models all allocation activity; we leave it to future work.

5.5. Boundaries of the Guarantee

The analysis above covers checked accesses in strict mode. Four boundaries delimit it.

Optimized checks. The proof obligation is for base instrumentation. Section 6 treats every optimization as a local replacement governed by a preservation rule: a check may be elided, merged, or hoisted only when LLVM analyses prove that the replacement covers the same accesses with compatible metadata, and PTSan falls back to the base check otherwise. Imprecision in those analyses therefore costs performance, not safety. ID-strip hoisting (Section 6.7) relocates tag strips and retags rather than checks; it is check-neutral because a retag reattaches the identifier recovered from the pointer's own derivation, so every check still consults the object named by the original ID.

Coverage. Where an allocation site is not intercepted, its objects are untagged (and fail closed in strict mode); where a memory access is not instrumented (the coverage boundaries of Section 5.1), it is unchecked. Protection degrades only for the specific untagged pointer or unchecked access, and many libc routines are covered by wrappers that validate pointer arguments before handing them to uninstrumented code. Identifier exhaustion is bounded the same way: when more than 2^{16} objects (2^{15} under Linear Address Masking) are live at once, strict mode fails closed on the next checked access through the fallback identifier, preserving safety at the cost of stopping the program. Section 8.3 reports identifier pressure empirically.

Concurrency. Spatial checks remain sound in multithreaded programs: each checked access validates its identifier and access range against the metadata it observes. Temporal checks are weaker under unsynchronized free/use races, because the bounds load and the check are not one atomic step with the deallocation, and the bounds load may be hoisted away from the access (Section 6). A racy use-after-free is a data race, a class of error outside PTSan's scope, and we treat it as outside the temporal guarantee. Non-racy temporal errors are unaffected by multithreaded programs.

Granularity. PTSan tracks allocation objects, not sub-objects, so intra-object field overflows are outside the spatial guarantee (Section 5). The evaluation measures this boundary directly (Section 8.4).

6. Optimizable Memory Safety

PTSan's base operations are cheap by construction: ID extraction is a single shift of a register value, the bounds lookup is a direct index into a flat table, and stack IDs come from thread-local state (Sections 4 and 7). This section is about the second half of the overhead story: because identity propagation is free and bounds loads are separable from the checks they feed, every remaining check component (ID extraction, bounds load, range computation, and branch) is ordinary LLVM IR that the compiler can optimize independently. The same holds for the software strips that produce canonical addresses (Section 4.1): they too are ordinary IR, and PTSan places them with a global cost model (Section 6.7).

This section describes those transformations, including the two-stage pipeline that optimizes loop checks, assuming the correctness of the LLVM analyses on which PTSan builds: dominance, postdominance, loop information, target library information, and Scalar Evolution. When an analysis cannot prove the condition an optimization requires, PTSan falls back to a less optimized check, so imprecision affects performance, not the base guarantee (Section 5).

6.1. Analysis Framework

PTSan first reduces each memory access to four facts: the base pointer from which the access is derived, the location where bounds may be loaded, the range of bytes that the access may touch, and the set of program points where a check for that access could be inserted.

Base-pointer analysis. For each access operand, PTSan traces the LLVM value from which the operand is derived, the access's *base pointer*, using LLVM's `getUnderlyingObject` plus handling for PHI nodes and selects: if all incoming values share a base, that base is used; otherwise the PHI/select itself is. The analysis is best effort, since the access pointer always remains a valid base.

Range analysis. PTSan represents each memory operation as a symbolic byte interval: the size comes from the LLVM type or the intrinsic's length operand, and the lower endpoint is the Scalar Evolution expression for the access pointer when it can be expanded safely at a candidate insertion point, or the access pointer itself otherwise. For loops, PTSan represents the whole loop as one range, from the first access through the last access plus the access width, when Scalar Evolution can compute that extent. These range objects are what the pass compares, merges, serializes during loop preanalysis, and finally materializes at the selected check location.

Placement analysis. A check insertion point must satisfy three conditions: it must dominate every access it covers; the bounds load must dominate it and the range must be expandable there; and the access must postdominate it, so a hoisted check cannot reject a path that would not have executed the access. When hoisting a check out of a loop, PTSan also requires the access to execute on every relevant iteration; otherwise a preheader check could reject an iteration in which the guarded access would not have run.

6.2. Local Preservation Rule

The optimizations below are not proven one at a time. Instead, they are all instances of a single local preservation rule, stated here so the per-transformation descriptions can simply point to the clause they rely on. Replacing a set of base checks with a new check is valid when all of the following hold:

1) The replacement check uses the same object ID as every check it covers.
2) The replacement range is the union of the covered access ranges.
3) The replacement check sits at a legal insertion point, in the sense of the placement analysis above, for the covered accesses. With one refinement: a *set* of covered accesses whose ranges union to the replacement range must postdominate the check. On any path where the check executes, those accesses also execute, so the original program would have checked at least the replacement range; a conditional access already covered by a guaranteed access need not postdominate the check on its own.
4) The replacement check is dominated by its bounds load, and no program point that may change the identifier's metadata (a free, resize, or unmap) lies between the bounds load and any covered access.

The remaining subsections instantiate this rule. ID-extraction hoisting moves only a pure extraction (the ID is the upper 16 bits of the SSA value) from each access to the dominating base pointer: in a loop, the low address of `buf[i]` changes every iteration but its ID does not, so repeated shifts and masks leave the loop body. Bounds-load hoisting enforces clause 4 by stopping at potentially freeing calls; check elision is the degenerate case where the replacement is no check; and check merging and loop hoisting replace several access ranges with one statically proven range covering their union, sharing a single bounds load so that clause 4 holds once for every covered access. One optimization sits outside the rule: ID-strip hoisting (Section 6.7) moves the compatibility strips and retags of Section 4.1 rather than checks, and its obligation, that every dereference still receives a canonical address and every check the identifier of the same source object, is stated there.

6.3. Hoisting Bounds Loads

Bounds loads are more constrained than ID extraction. IDs are part of SSA values and do not change, but bounds metadata can change when an object is freed, resized, or unmapped. PTSan therefore hoists a bounds load only to a point where the metadata cannot be invalidated on any path to the checked access, as required by clause 4 of the preservation rule.

For unfreeable objects, such as stack allocations within their function lifetime and globals, PTSan can load bounds

near the base pointer definition. For heap and mmap objects, the pass treats unknown calls as potential lifetime changes. It may hoist across LLVM intrinsics and calls that LLVM marks `nofree`, and it treats nonreturning calls as not reaching the checked access. Otherwise, the nearest potentially freeing call blocks hoisting. In control-flow regions with multiple paths, PTSan searches for the highest insertion point that dominates the access and from which no blocking call is reachable before the access.

This rule is about temporal precision: spatially, loading an older bound for the same object cannot let an access escape that object's allocation interval, but a load hoisted across a free could miss an intervening lifetime end. The rule preserves the base temporal property in the absence of data races; racy free/use executions are outside the temporal guarantee (Section 5.5).

For stack and global objects, PTSan eliminates the bounds load entirely: when the base pointer is an `alloca` or a global, the base and size are known statically, so the pass folds both into the check as constants and the ID extraction and metadata loads disappear, leaving only the interval comparison. This is the limit case of clause 4, since the metadata for these objects cannot change within their lifetime.

6.4. Conservative Check Elision

PTSan eliminates a check only when enforcement cannot need it. Zero-size checks are removed, since the runtime treats them as vacuously safe. Checks are also removed for accesses that LLVM proves dereferenceable and aligned, but only when the base object is unfreeable (stack objects, constants, and supported global-derived pointers): eliding a provably in-bounds *heap* check could remove the detection of a use-after-free before ID reuse, so the unfreeable condition keeps elision correctness-preserving rather than a spatial-only shortcut.

6.5. Merging Checks

Many programs perform several nearby accesses through the same object ID. PTSan merges such checks when one replacement range can cover all of the original accesses at a legal insertion point.

Subsuming merge. If two checks use the same base pointer and the same bounds load, and one access range contains the other, the larger range can cover both accesses. The merged check is placed at a point that dominates the covered accesses and is postdominated by them according to the placement analysis above. This case is common in load/store pairs.

Orderable merge. If PTSan can prove that two ranges are ordered, it may replace them with one interval from the lower range start to the upper range end. This covers unrolled sequences such as `buf[i]` through `buf[i+3]` with one combined range. The comparison and merge happen at compile time using Scalar Evolution; PTSan never adds runtime tests to decide whether checks can merge.

Both merge strategies require the merged checks to share a bounds load: checks are grouped by base pointer and bounds-load location, which establishes clause 4 for the merged check. No covered access can observe metadata loaded under different lifetime assumptions, because every covered access is checked against the same load and no metadata-changing point separates that load from the access.

6.6. Loop Checks

A local sanitizer check in a hot loop adds arithmetic, loads, and a branch to every iteration. PTSan uses Scalar Evolution to replace such checks with one preheader check when it can compute the full range touched by the loop.

For an access whose address is represented by a Scalar Evolution add-recurrence, PTSan asks LLVM for the first value, the last value, and the loop trip count. If the access moves monotonically upward, the loop range starts at the first access and extends through the last access plus the access width. If it moves monotonically downward, the endpoints are swapped. If the step is loop-invariant but the sign is not statically known, PTSan materializes a loop-range check that selects the minimum endpoint at runtime and checks the absolute span. If the step is not analyzable, the pass leaves the check in the loop.

Hoisting also requires the preheader to be a legal insertion point, including the every-iteration condition of the placement analysis (Section 6.1). These conditions constrain only the range *check*: the bounds load it depends on can be hoisted much more freely, under only the metadata-stability conditions of Section 6.3. Decoupling the bounds load from the check this way is something most sanitizers do not do.

These hoisting conditions fail on exactly the loops that matter most. After LLVM vectorizes a loop, its body holds alternative access sequences, a vector fast path and a scalar remainder, and neither postdominates the loop body, so most optimized hot loops cannot be hoisted after the fact. Instrumenting before vectorization would restore the simple scalar structure but disrupt the loop optimizations themselves. PTSan therefore uses a two-stage pipeline: it analyzes loop accesses early, while they are still in scalar form, preserves the results as metadata, and consumes them after LLVM has transformed the loop.

The early pass does not modify program instructions. It finds loop accesses that can be described by base pointer and access range, records whether an entire loop or an entire base within a loop has been solved, and serializes the result into loop metadata. Later, the final PTSan instrumentation pass consumes that metadata. If the metadata says that a loop is completely covered, the pass can ignore the individual memory operations that remain inside the transformed loop and emit the precomputed range checks instead. If only a particular base was solved, PTSan elides checks for accesses derived from that base and falls back to local analysis for the rest.

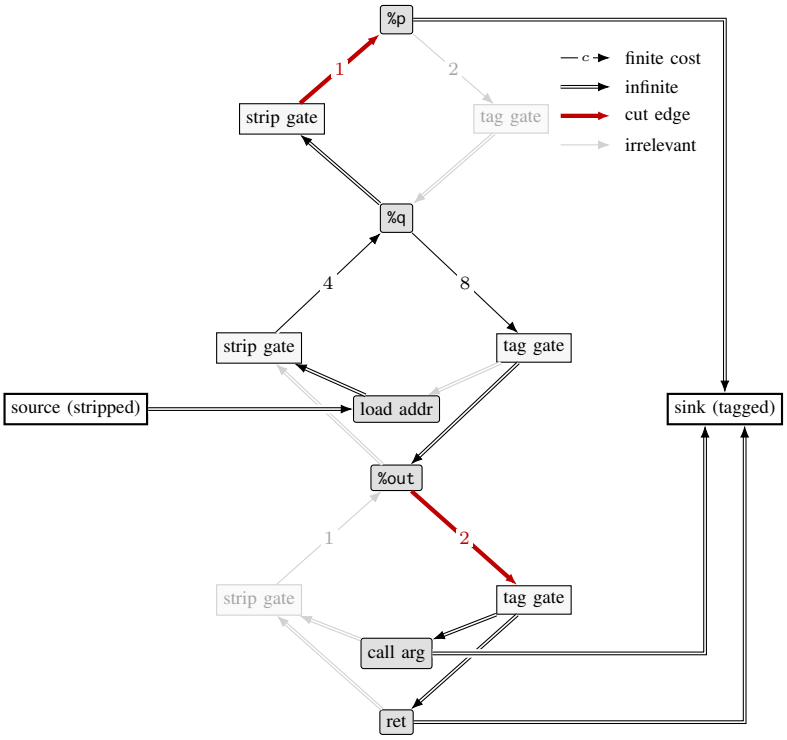
The implementation also preserves PTSan loop metadata across LLVM loop transforms that clone or restructure loops, including vectorization, unrolling, runtime unrolling,

```
f(%p):
 entry:
  br loop
 loop:
  %q = phi [%p, entry], [%next, loop]
  %s = strip %q
  %v = load %s
  %next = gep %q, 1
  br ..., loop, exit
 exit:
  %out = phi [%q, loop]
  call @g(%out)
  ret %out
```

(a) Base placement: strip %q in the loop.

```
f(%p):
 entry:
  %p.id = get_id %p
  %p.s = strip %p
  br loop
 loop:
  %q.s = phi [%p.s, entry], [%next.s, loop]
  %v = load %q.s
  %next.s = gep %q.s, 1
  br ..., loop, exit
 exit:
  %out.s = phi [%q.s, loop]
  %out.t = tag %out.s, %p.id
  call @g(%out.t)
  ret %out.t
```

(b) Hoisted: strip %p, retag %out.



(c) Minimum-cut solution; irrelevant edges are faded.

Figure 2. ID-strip hoisting. Finite labels are capacities: strip and tag cost 1 and 2, respectively, multiplied by 4 per loop level. The base placement in (a) cuts %q's strip-cost edge (4). The minimum cut (red) instead cuts %p's strip-cost edge (1) and %out's tag-cost edge (2), producing (b) with one retag shared by both escaping uses.

unroll-and-jam, and loop distribution. This metadata propagation is an engineering detail, but it is central to the performance result: PTSan can let LLVM optimize the program first and still recover the loop-level memory ranges needed for low-overhead checking.

6.7. Hoisting ID Strips

The optimizations above reduce the cost of checking; ID-strip hoisting reduces the cost of compatibility. Without LAM, every dereference requires a canonical address (Section 4.1), so base instrumentation strips the identifier with one bitwise mask, immediately before each memory operation. Each strip is cheap, but a strip inside a hot loop executes on every iteration; unhoisted stripping accounts for roughly 20 percentage points of PTSan's SPEC CPU 2017 overhead (Section 8.5).

Figure 2 shows ID-strip hoisting before and after. In Figure 2a, the original base placement strips the loop-varying address %q on every iteration. In Figure 2b, the strip has been hoisted through the address computation to the loop-invariant base %p and executes once, in the entry block; the address the loop loads from, %q.s derived from %p.s, is already canonical. The price appears after the loop: %out is now stripped but escapes at the call and the return, where PTSan requires tagged values, so a *retag* must reattach the identifier saved at entry. Here one retag serves both escaping uses, and the exchange is a clear win once the loop runs more than a few iterations. Blind ID-strip hoisting is not necessarily a win in all circumstances: a retag forced *inside* a loop can cost more than the strips it saves. Strip placement is therefore a global cost problem rather than a peephole rule, and the ablation bears this out: on SPEC, a local placement rule recovers less than half of what the global formulation below does (Section 8.5).

PTSan formulates the problem over polarities. Each pointer-typed SSA value is assigned a *polarity*, stripped or tagged, subject to two constraint sets. Uses are constrained individually: dereference address operands need stripped values, while identifier extractions feeding checks, pointer-to-integer casts, and each point where a pointer escapes local dataflow (return values, call arguments, and pointer values stored to memory) need tagged values. Values are constrained by recoverability: only pass-through definitions (getelementptr address computations, PHI nodes, selects, and similar) may lose their tagged version. This is because a retag can recover their identifier from a dominating tagged ancestor, as the retag of Figure 2b recovers %out's identifier from %p. For a PHI or select, PTSan synthesizes a PHI or select over the ancestors' identifiers. Values that originate pointers, such as arguments, call results, and loaded pointer values, stay tagged: stripping them would discard the only copy of the identifier.

Choosing polarities is then a two-terminal minimum cut: a source representing stripped, a sink representing tagged, a node per SSA value, and the constraints above as infinite

```
procedure BUILDGRAPH(F)
    G ← graph containing each pointer value in F
    S_strip, T_tag ← NEWNODE(G), NEWNODE(G)
    for all pointer-valued SSA definitions v in F do
        if v originates a pointer then
            EDGE(v, T_tag, ∞)
            continue
        if v is a PHI or select over pointer inputs X then
            for all x ∈ X do
                ENSUREGATES(G, x)
                EDGE(T(x), v, ∞)
                EDGE(v, S(x), ∞)
            continue
        x ← pointer operand of v
        ENSUREGATES(G, x)
        if v is TAGGED(x) then
            EDGE(v, T_tag, ∞)
            EDGE(T(x), v, ∞)
            EDGE(v, S(x), ∞)
        else if v is STRIP(x) then
            EDGE(S_strip, v, ∞)
            EDGE(S_strip, S(x), ∞)
        else if v is transparent with pointer operand x then
            EDGE(T(x), v, ∞)
            EDGE(v, S(x), ∞)
    return G
procedure ENSUREGATES(G, v)
    if v has no conversion gates then
        T(v), S(v) ← NEWNODE(G), NEWNODE(G)
        EDGE(v, T(v), TAGCOST(v))
        EDGE(S(v), v, STRIPCOST(v))
```

Figure 3. Construction of the polarity min-cut graph. Pointer SSA values are graph nodes, and $T(v)$ and $S(v)$ are value v's shared tag and strip conversion gates. Hard consumers have already been made explicit as TAGGED and STRIP SSA anchors. Conversion costs include loop weighting.

terminal edges (need-stripped uses tie to the source; need-tagged uses and pointer-originating values tie to the sink). The subtlety is how to charge for conversions. Connecting each value directly to each of its users would charge per boundary-crossing use: a stripped `%out` in Figure 2a would pay for two retags, although the rewrite materializes one. PTSan instead routes every use through two per-value *gate* nodes representing the option of materializing one tagged or one stripped version of the value; in Figure 2c, each value owns such a pair. The use edges are infinite-cost, so a cut can separate a value from differently-polarized uses only by crossing a finite gate edge, and however many uses share that conversion, the cut pays its cost once: the minimum cut prices exactly the conversions the rewrite will insert. Figure 3 summarizes this construction. Figure 2c assembles the whole instance for the example function and shows its minimum cut: the terminal ties force the load's address use to the source side and the call use, the return use, and the argument `%p` to the sink side, and the cut crosses exactly two finite edges, the strip of `%p` and the retag of `%out`. (Only edges from the source side to the sink side count in a directed cut, so the infinite edges that cross back are free.) Edge weights grow exponentially with loop depth, so conversions migrate out of loop nests, and retags weigh more than strips because a retag also synthesizes its identifier and holds it live, adding register pressure. PTSan computes the cut with Dinic's max-flow algorithm [20], reads each value's polarity off its side of the cut, inserts the strips and retags the cut edges denote, and removes conversions the assignment makes redundant.

The pass runs late, after check ranges have been materialized as ordinary IR, so every producer and consumer of tagged values is visible to the graph. It is guarantee-neutral by construction: check operands are need-tagged uses and pointer-originating values keep every identifier recoverable, so hoisting changes where conversions execute, never which object a check consults. Under LAM, dereferences leave the need-stripped set, and the same analysis operates on the few strips that remain at canonical-address boundaries such as inline assembly and kernel-bound pointers.

7. Implementation

PTSan is implemented as an LLVM 19 sanitizer: an application is rebuilt with `-fsanitize=ptsan`, with no source changes; the flag runs PTSan's passes and links the static compiler-rt runtime. PTSan targets x86-64 and AArch64: the passes operate on LLVM IR and are architecture-independent.

Two passes implement the pipeline of Section 6: the loop preanalysis pass runs at the vectorizer-start extension point and writes only metadata, and the main instrumentation pass allocates IDs, inserts checks, strips IDs where pointers escape to code expecting raw addresses (placing strips and retags with the min-cut analysis of Section 6.7), and releases stack IDs on function exit. The runtime's metadata tables are parameterized by the ID width (16 bits by default; the LAM build uses 15 and enables LAM_U48 at startup) and are backed by roughly 950 libc interceptors built on LLVM's sanitizer interception infrastructure: allocation and lifetime wrappers cover the `malloc` family, `mmap`, and `munmap`; generic wrappers strip pointer arguments and retag returned pointers that point into an argument; and string, memory, `pthread_create`, and formatting functions have hand-written support. The remaining compatibility limits are discussed in Section 5.5. The implementation is about 13,700 lines of code across the passes and runtime, plus 7,700 lines of tests. PTSan, comprising the modified LLVM toolchain and the compiler-rt runtime, is available as open source at `https://github.com/Aarno-Labs/PointerTagSanitizer`. We intend to propose PTSan for upstream inclusion in LLVM: following the structure of sanitizers (instrumentation passes, a compiler-rt runtime, and the shared interceptor infrastructure) is a deliberate step toward that goal.

8. Evaluation

This section presents results that test whether PTSan delivers pointer-based memory safety at a deployable cost. Our results support four claims:

1) On stock x86-64 hardware and kernels, PTSan's runtime overhead is 57.2% geomean on SPEC CPU 2017 and 31.5% on LLVM MultiSource, at parity with the fastest published redzone sanitizer and roughly a third of the

published overhead of prior pointer-based systems, and hardware address masking reduces these to 46.4% and 27.2%.
2) The cost is stable across architectures: the same suites measure 54.7% and 40.8% on ARM64 without hardware address masking.
3) PTSan's physical memory overhead is 1.014× geomean, effectively native, where ASan and RSan cost 3.33× and 3.01×.
4) On MSET, PTSan matches the originally published detection coverage of SoftBound+CETS, the strongest prior pointer-based system, including categories that no location-based sanitizer detects; only SoftBound+CETS's revisited port does better, and only on the intra-object cases which PTSan does not target.

8.1. Experimental Setup

Hardware and toolchain x86-64 runtime-overhead experiments (SPEC, LLVM MultiSource, the ablation, and the server workloads) run on an Intel Core Ultra 9 285H machine with 96 GB of memory and Ubuntu 24.04. ARM64 experiments run on an AWS m7g.2xlarge instance: an AWS Graviton3 (Arm Neoverse-V1, eight physical cores, no SMT, fixed 2.6 GHz clock) with 32 GiB of memory and Ubuntu 24.04, with each benchmark pinned to a single core and reported as the median of three runs. All native and instrumented builds use `-O3`. The memory overhead, RSan, and MSET experiments run on a separate AMD Ryzen 9 9950X3D machine with 60 GiB of memory and Ubuntu 24.04, with each benchmark pinned to a fixed physical core under the `performance` governor.

Configurations We report three configurations, named by architecture. The default, *PTSan-x86*, enables all the optimizations of Section 6, strips identifier bits in software where uninstrumented code requires canonical addresses, and supports 2^{16} live object IDs; it runs on stock hardware and kernels.

PTSan-LAM instead uses Intel Linear Address Masking to elide most stripping, with a 2^{15} ID budget. We report PTSan-LAM as a secondary configuration rather than the headline because of its deployment caveats. PTSan needs the LAM_U48 variant [11], which is not in mainline Linux: the mainlined LAM_U57 exposes too few maskable bits for PTSan's ID budget, and address masking has drawn security concerns since SLAM showed it enlarges the Spectre gadget surface [21]. Our measurements use a custom Linux 6.1 kernel. As address-masking hardware and kernel support mature, the PTSan-LAM numbers indicate where PTSan's overhead is headed.

PTSan-ARM is the PTSan-x86 configuration compiled for AArch64, where stripping is always in software. Where the architecture does not matter, we write simply PTSan. All runtime-overhead measurements use PTSan's default stack-ID handling: the thread-local fast path reuses stack identifiers immediately and does not clear their bounds entries at function exit.

We compare against AddressSanitizer (ASan) [7] and RangeSanitizer (RSan) [8] measured locally, and against the published overheads of SoftBound+CETS [13, 14] and CUP [10]. Following its upstream methodology, RSan is measured against its own `-O2` baseline built with the same RSan LLVM build; its runs cover SPEC's pure-C/C++ rate benchmarks (excluding the mixed-language `511.povray_r`, `526.blender_r`, and `507.cactuBSSN_r`) minus `502.gcc_r`, whose RSan build does not complete, leaving a 13-benchmark subset. Each system is therefore reported as overhead over its own baseline, which removes compiler and machine differences from the comparison.

Benchmarks Our CPU evaluation uses two suites. First, we run SPEC CPU 2017 v1.1.9 `intrate` and `fprate` with reference inputs, base tuning, and one copy; we evaluate the 17 C/C++ benchmarks and exclude benchmarks that execute Fortran code. Second, we run the 167 LLVM test-suite MultiSource benchmarks, which the LLVM documentation describes as "entire programs with multiple source files" where "large benchmarks and whole applications go" [22]. We exclude MultiSource programs with native runtime under 10 ms, which are too noise-sensitive to measure, and five programs whose runs PTSan flags: in four (`gs`, `netbench-url`, `hbd`, and `PENNANT`) PTSan detects real memory-safety bugs[2], and `siod` walks the stack across frames by address arithmetic, which violates PTSan's pointer-authority model (Section 5) without being a memory-safety bug. Because flagged runs take PTSan's intentionally slow report path, they would skew the overhead numbers; 149 programs remain on x86-64 and 148 on ARM64, where `security-sha` also falls under the runtime cutoff. Overheads are reported as the geometric mean of per-benchmark instrumented/native runtime ratios, always over the same benchmark set within a comparison. For memory overhead we report peak physical memory from cgroup v2 `memory.peak`, not virtual reservation, taking the maximum successful command row per benchmark. For detection coverage we use MSET [12], described with its detection convention in Section 8.4.

8.2. Runtime Overhead

PTSan provides pointer-based checking at 57.2% geomean overhead on SPEC CPU 2017 on commodity x86-64 systems and 54.7% on ARM64, at parity with the fastest redzone sanitizer. ID-strip hoisting (Section 6.7) recovers most of the software tag-stripping cost on stock hardware; hardware address masking removes what remains, bringing the x86-64 geomean to 46.4%.

8.2.1. SPEC CPU 2017. Table 2 reports per-benchmark overhead for all three configurations. PTSan-x86's geomean overhead is 70.6% on the integer suite and 43.3% on floating

2. We root-caused these violations to assess disclosure obligations and found no undisclosed vulnerability in maintained software: the `gs` and `PENNANT` bugs are already fixed in the maintained upstreams (Ghostscript and LANL PENNANT), and `netbench-url` and `hbd` exist only in the public benchmark corpus, with no maintained upstream.

TABLE 2. RUNTIME OVERHEAD AND PEAK LIVE OBJECTS ON SPEC CPU 2017 (†: EXCEEDS THE 2^{16} ID BUDGET).

Benchmark	PTSan-x86	PTSan-LAM	PTSan-ARM	Max live IDs
SPECrate 2017 Integer				
500.perlbench_r	181.4%	117.2%	131.1%	1,253,094†
502.gcc_r	109.4%	61.4%	95.4%	1,076,541†
505.mcf_r	36.2%	32.3%	49.6%	15,010
520.omnetpp_r	73.5%	61.8%	17.0%	2,388,303†
523.xalancbmk_r	75.8%	57.7%	76.7%	2,338,108†
525.x264_r	44.8%	40.6%	75.2%	3,498
531.deepsjeng_r	56.9%	49.8%	72.5%	650
541.leela_r	64.2%	57.3%	125.8%	26,002
557.xz_r	34.0%	31.9%	-2.8%	42
Geomean (INT)	**70.6%**	**55.0%**	**65.4%**	
SPECrate 2017 Floating Point				
507.cactuBSSN_r	57.2%	57.2%	35.9%	18,655
508.namd_r	73.9%	62.9%	34.1%	3,492
510.parest_r	39.0%	39.3%	44.4%	2,375,107†
511.povray_r	136.5%	118.3%	170.6%	11,869
519.lbm_r	-2.1%	-3.1%	-12.8%	2
526.blender_r	18.0%	8.8%	37.8%	342,174†
538.imagick_r	45.4%	36.7%	54.9%	480
544.nab_r	17.9%	12.4%	36.0%	80,183†
Geomean (FP)	**43.3%**	**37.3%**	**43.5%**	
Geomean (combined)	**57.2%**	**46.4%**	**54.7%**	

point, for a combined 57.2%. Every benchmark builds, runs, and validates on both architectures; the worst cases are `500.perlbench_r` at 181% on x86-64 and `511.povray_r` at 171% on ARM64.

The outliers have a consistent shape. For `500.perlbench_r`, profiling attributes the cost to inline-check instruction growth rather than runtime-library time or locality effects: instrumentation grows Perl's regex interpreter `S_regmatch` about 9× in code size, that one function dominates PTSan execution samples while runtime helpers stay under 2%, and IPC rises. Large interpreters over pointer-rich state are close to a worst case for any pointer-based instrumentation. `511.povray_r`'s overhead (136% on x86-64) instead concentrates in stack-object ID setup for its many small, hot functions with address-taken temporaries, consistent with the stack-ID rung of the ablation (Section 8.5).

PTSan-LAM isolates the software tag-stripping cost that remains after ID-strip hoisting: with stripping elided in hardware, the combined geomean falls from 57.2% to 46.4%. Of the roughly 20-point gap between the pipeline without strip hoisting (66.2%, Section 8.5) and PTSan-LAM, the compiler now recovers about half in software, which validates the design decision to keep stripping separable and optimizable (Section 6).

PTSan-ARM shows the cost is portable. The pipeline compiles unmodified for AArch64 (all PTSan operations are IR-level, so the only difference is in the LLVM-backend-generated assembly), and its combined geomean is 54.7% against PTSan-x86's 57.2%. Per-benchmark costs diverge more than the aggregates suggest, and while we did not profile the ARM64 runs to attribute them individually, the pattern tracks per-check ISA cost. The x86-64 check sequence is compact: the bounds-table access folds into one complex-addressing load, and the compare and branch macro-fuse; AArch64 needs a few more discrete instructions per check. Call-dense, check-dense, recursion-shaped code amplifies

TABLE 3. OVERHEAD ON LLVM MULTISOURCE; THE ID-BUDGET ROW COUNTS ALL 167 PROGRAMS.

	PTSan-x86	PTSan-LAM	PTSan-ARM
Programs measured	149	149	148
Geomean overhead	31.5%	27.2%	40.8%
Median overhead	24.7%	21.2%	34.9%
Programs under 50% overhead	103 (69%)	114 (77%)	84 (57%)
Within ID budget (2^{16} / 2^{15} / 2^{16})	152/167 (91%)	149/167 (89%)	152/167 (91%)

that difference and loop-shaped code hides it: `541.leela_r` (recursive tree search) and `511.povray_r` (stack-ID-heavy) degrade on ARM64, while streaming floating-point codes such as `508.namd_r` run cheaper there.

For context, ASan's combined geomean on the same machine is 117.4%, roughly double PTSan-x86's overhead and 2.5× PTSan-LAM's. RSan, the fastest published redzone-style sanitizer, reports 44.5% on SPEC CPU 2017 speed with its default implicit-tagging design [8]; our local runs of that design measure 50.6% against its own baseline on its 13-benchmark feasible subset. On that subset PTSan-x86's geomean is 52.3% and PTSan-LAM's is 43.4%: on stock hardware PTSan runs at parity with RSan, and with hardware masking it is faster. That comparison gives RSan no hardware disadvantage: RSan's own address-masking experiments show that explicit tagging on Intel LAM runs slightly *slower* than its implicit default (54.0% vs. 51.0% on SPEC CPU 2006), masking instead reducing its memory overhead [8].

Pointer-based checking no longer trades speed for its guarantee: object-identity detection that no location-based sanitizer provides (Section 8.4) and 3× lower memory footprint (Section 8.3) now come at redzone-sanitizer speed. The published overheads of the pointer-based systems with comparable guarantees, 161% for SoftBound+CETS and 158% for CUP, are roughly triple PTSan's default configuration; those numbers come from older SPEC suites, so we treat them as indicative rather than directly comparable.

8.2.2. LLVM MultiSource. The MultiSource suite tests breadth: a large set of C and C++ programs across many application domains rather than a tuned benchmark kernel set. Table 3 summarizes the results; the distribution appears in Figure 4 (Appendix A). PTSan-x86's geomean overhead is 31.5% (27.2% with LAM, 40.8% on ARM64). The median, 24.7%, is below the geomean, and 69% of programs stay under 50% overhead, so the aggregate is not driven by a few favorable outliers. The tail is the same pointer-dense shape as the SPEC outliers (`kimwitu++`, `sqlite3`). On ARM64 the tail shifts, led by `lambda` (534%) and `sqlite3` (253%). `lambda`, a small lambda-calculus interpreter, is the extreme of the recursion-shaped pattern above: it executes 3.8 billion checks, nearly all single-access rather than hoistable loop-range checks, in a roughly two-second run, a dependent pointer-chasing chain in which checks are the majority of the dynamic instruction stream, leaving no slack to hide the added per-check instructions on AArch64. SPEC overheads are higher than MultiSource overheads because the SPEC kernels spend more of their time in long, pointer-dense

hot loops; the MultiSource programs look more like the application code a sanitizer would actually be deployed on.

8.2.3. Real-World Servers. To test deployment-style workloads, we ran PTSan on seven real-world server and cryptographic workloads from the Phoronix Test Suite [23], each in its default configuration, on the Intel machine of Section 8.1 (Table 11; per-workload descriptions in Appendix A.1). Overhead tracks how compute-bound each workload is: Apache is within measurement noise, and OpenSSL, SQLite, and memcached stay at or below 9% with PTSan-x86, while the more allocation- and request-processing-intensive PostgreSQL, Redis, and NGINX reach 16%, 26%, and 44%, respectively. Hardware masking's benefit is uneven: it nearly eliminates NGINX's overhead (44% to 2%), whose cost is dominated by tag stripping, but barely changes Redis's, leaving a PTSan-LAM maximum of 25%. Peak live-ID demand never exceeds 16,411 (Redis), about a quarter of the 2^{16} budget, so no workload approaches exhaustion. These results demonstrate PTSan's applicability to real-world server workloads at deployable overhead.

8.3. Memory Overhead

PTSan's physical-memory overhead on SPEC is 1.014× geomean with a maximum of 1.121×: effectively native. On the same benchmarks and baseline, ASan costs 3.33× and RSan 3.01× (per-benchmark data in Appendix A). Restricting the comparison to benchmarks within PTSan's 2^{16} identifier budget does not change the story: on the 8 in-budget benchmarks with memory data (the 13-benchmark memory subset lacks `507.cactuBSSN_r` and `511.povray_r`, so 8 of the 10 in Table 2 qualify), RSan costs 2.37× where PTSan costs 1.021×, so the advantage is not an artifact of workloads whose identifier demand PTSan cannot cover. As memory becomes a first-order cost in production fleets, near-native memory is the difference between a sanitizer that can ship and one that cannot.

The contrast is sharpest on small-footprint programs: `541.leela_r` has a 31 MiB native peak, which redzone and shadow-memory designs inflate to over a gigabyte (34–37×) while PTSan reaches 1.121×. Redzone and shadow costs scale with the number and size of allocations; PTSan has no redzones, no shadow memory, no quarantine, and no per-pointer metadata in memory. Its metadata is a fixed, workload-independent set of object-ID tables: about 1.2 MiB at the default 2^{16} budget and a few MiB including the shared sanitizer interceptor state. The overhead is therefore bounded by design rather than scaling with allocations.

8.3.1. ID Pressure. The flip side of fixed metadata is a finite ID budget, so we measure each workload's peak demand for simultaneously live IDs. On SPEC, 10 of 17 benchmarks stay within the default 2^{16} budget (Table 2); the 7 that exceed it (e.g., `520.omnetpp_r` at 2.4M live IDs) keep millions of small allocations live concurrently. SPEC overstates this pressure for at least `500.perlbench_r` and `502.gcc_r`, where the SPEC harness disables garbage collection, so these numbers are not necessarily indicative of real-world deployments of those programs. On LLVM MultiSource, 152 of 167 programs (91.0%) fit the 2^{16} budget and 149 (89.2%) fit the 2^{15} LAM budget. When demand exceeds the budget, PTSan falls back to shared IDs and protection degrades for the affected objects (Section 4.3); strict deployments can instead fail closed. These results indicate that the compact-ID design covers most workloads outright; because peak ID demand is cheap to measure, a deployment can determine in advance whether a workload fits the budget rather than discovering exhaustion in production.

TABLE 4. MSET DETECTION RESULTS. ASAN, RSAN, AND PTSAN ARE LOCAL RUNS; THE SOFTBOUND+CETS ROWS ARE PUBLISHED RESULTS, FOR THE ORIGINAL SYSTEM AND ITS REVISITED LLVM PORT.

System	Linear	Non-linear	Type conf.	UAF/UAR	Double-free	Misuse-free
ASan	70/90	18/72	18/30	8/16	4/4	20/20
RSan	58/90	18/72	16/30	10/16	4/4	0/20
SoftBound+CETS [13, 14]	72/90	54/72	24/30	16/16	4/4	20/20
SoftBound+CETS revisited [9]	84/90	66/72	28/30	16/16	4/4	20/20
PTSan	72/90	54/72	24/30	16/16	4/4	20/20

8.4. Detection Coverage

On MSET, PTSan detects every inter-object spatial, non-object spatial, and temporal case[3], matching the originally published coverage of SoftBound+CETS, the strongest prior pointer-based system, at a fraction of its overhead.

MSET separates ordinary address-validity failures from object-identity failures: an access can land inside mapped, currently allocated memory and still be illegal for the pointer that performed it [12]. Table 4 reports the stock bug-family aggregates under the MSET detection convention (a test counts as detected if the sanitizer reports the violation or prevents the benchmark's preconditions from being established), which is also the convention of the published SoftBound+CETS rows.

In the relation-bucket view, PTSan detects 126/126 inter-object and 24/24 non-object spatial cases, and all temporal cases: 16/16 use-after, 4/4 double-free, and 20/20 misuse-of-free, with temporal detection deterministic until the corresponding ID is reused (Section 5.4). The 0/42 intra-object result is by design and matches the scope set in Section 5.5. Because PTSan detects every inter-object and non-object case, all of its spatial misses in Table 4 (18 linear, 18 non-linear, and 6 type-confusion, 42 in total) are exactly these 42 intra-object cases. The only row that exceeds PTSan is the revisited SoftBound+CETS port, and its entire advantage is 28 of those 42 intra-object cases, recovered by narrowing bounds at subobject derivations, a mechanism PTSan does not attempt.

On non-linear out-of-bounds accesses, the category that exercises redirection into another live object (the same pattern as the ImageMagick example of Section 2), PTSan detects all supported cases (the undetected cases are intra-object out-of-bounds accesses), while ASan and RSan detect

3. In order to catch stack temporal errors, PTSan needs to be run with `-ptsan-stack-temporal-safety`, which adds ∼3% overhead on SPEC CPU 2017 (Table 10).

none (see Table 7 in Appendix A for supported-only data). This is a semantic gap, not an implementation gap: a range or address-validity check cannot reject an access whose target lies inside some other live object; only a pointer-identity check can. The temporal categories show a similar separation. Our detection evaluation is benchmark-based; an in-the-wild bug-finding study is future work.

8.5. Optimization Ablation

Table 5 isolates the contributions of the mechanisms of Section 6 as a cumulative waterfall, measured on the Intel machine of Section 8.1. The first row is the full pipeline with LAM (46.4%); each subsequent row disables one more mechanism than the row above, down to unoptimized base instrumentation (139.9%), so each Δ is the marginal cost of losing that mechanism given that everything above it in the table is already gone. Read bottom-up, the compiler optimizations together cut PTSan-x86's overhead from 140% to 57%, a $\sim$60% reduction.

Four effects account for most of the waterfall. ID-strip hoisting (Section 6.7) is worth 9 points on stock hardware. Most of that benefit, 5.3 points, comes from global rather than local strip placement, providing the empirical case for the global min-cut formulation. The stack-ID fast path is a runtime mechanism rather than an optimization pass; its rung (+13.6 points) is consistent with `511.povray_r`'s stack-heavy profile. Check merging adds 21.1 points. The final no-optimization rung is the largest (+27.9 points), as ID extraction can no longer be shared across derived pointers or hoisted out of loops. Because the waterfall is cumulative, mechanisms that overlap show small marginal deltas once their partner is gone: loop check hoisting adds only half a point after loop preanalysis has already been removed, although the two together account for several points. Per-benchmark data appears in Appendix A.

TABLE 5. OPTIMIZATION ABLATION ON SPEC CPU 2017: A CUMULATIVE WATERFALL IN WHICH EACH ROW DISABLES ONE MORE MECHANISM THAN THE ROW ABOVE IT; Δ IS THE MARGINAL COST IN PERCENTAGE POINTS OF COMBINED GEOMEAN OVERHEAD.

Configuration	INT	FP	Combined	Δ
Full pipeline (PTSan-LAM)	**55.0%**	**37.3%**	**46.4%**	
− hardware masking (PTSan-x86)	70.6%	43.3%	57.2%	+10.8
− global strip placement (local only)	75.2%	49.3%	62.5%	+5.3
− ID-strip hoisting entirely	80.2%	51.7%	66.2%	+3.7
− stack-ID fast path	87.8%	71.2%	79.8%	+13.6
− loop preanalysis	89.3%	82.3%	86.0%	+6.2
− loop check hoisting	90.0%	82.6%	86.5%	+0.5
− check merging	104.6%	111.1%	107.6%	+21.1
− check elision	107.5%	111.1%	109.2%	+1.6
− bounds-load hoisting	107.3%	117.5%	112.0%	+2.8
− ID-extraction hoisting (no opts.)	125.2%	157.7%	139.9%	+27.9

8.6. Summary

PTSan provides pointer-object authority checking with 57.2% runtime overhead on commodity x86-64 systems, 54.7% on ARM64, and 46.4% with LAM, 1.014$\times$ memory overhead, and full inter-object and temporal MSET coverage, in a recompile-only deployment model. PTSan's runtime overhead matches the best address-validity system on stock hardware while prior identity-based systems are roughly triple PTSan's runtime overhead percentage. No published system occupies this point.

9. Related Work

Surveys of the sanitizer space divide memory-safety checkers into *location-based* tools, which track which memory is valid, and *identity-based* tools, which track which object each pointer is entitled to access, its *intended referent* [4, 12]. Identity-based tools subdivide by where bounds live: *per-pointer* tracking attaches bounds to every pointer value, while *per-object* tracking keeps one bounds record per allocation and recovers it from the pointer. PTSan is an identity-based, per-object sanitizer that recovers metadata through a pointer-carried identifier, with temporal protection by identifier invalidation. Table 8 (Appendix A) summarizes guarantees and costs across both classes.

Location-Based Sanitizers. AddressSanitizer (ASan) popularized the practical sanitizer model: compile-time instrumentation, a runtime with allocator integration, shadow metadata, and high compatibility [7]. Its guarantee is adjacency: redzones detect accesses that land in the poisoned padding next to an object, so the P2 primitive of Section 3, an offset that jumps over the redzone into another live object, is not flagged. MSET quantifies the gap: ASan detects none of the non-linear out-of-bounds cases and misses half of the use-after-free cases, since its quarantine only delays reuse [12]. The cost profile for ASan is moderate runtime overhead, but severalfold memory growth from the redzones, shadow memory, and quarantine [4]. RangeSanitizer (RSan) is the strongest current sanitizer in this class: it encodes a size class in unused pointer bits so one metadata load and one comparison validate an entire access range, sharply cutting runtime overhead [8]. RSan derives temporal detection from the same check, zeroing an object's stored bound at free so a stale access fails; this catches use-after-free and double-free, but not invalid-free, because RSan never validates the freed pointer the way ASan's allocator does (0/20 versus 20/20 in our MSET runs). PTSan answers the provenance question ASan and RSan cannot, detects temporal errors after quarantine-scale reuse windows, and keeps memory near native, with no redzones, shadow memory, or address-space quarantine, at runtime cost matching RSan's (Section 8.2).

Per-Pointer Bounds Tracking. SoftBound attaches disjoint base and bound metadata to each pointer and narrows bounds at subobject derivations, which makes per-pointer tracking the only technique with the conceptual potential for complete spatial safety, including intra-object overflows [13, 12]. CETS adds a lock-and-key scheme with never-reused 64-bit keys, giving deterministic temporal detection even after memory reuse [14]. These are the strongest software guarantees in the design space, and PTSan does not match

them: it tracks allocation-granular bounds, so intra-object overflows are out of scope (Section 5), and its 16-bit identifiers are reused, so temporal detection is probabilistic (Section 5.4).

The cost of the stronger guarantee is what PTSan is designed to avoid. Every pointer copy, call, and store must propagate disjoint metadata through shadow structures, which keeps even a modern LLVM port of SoftBound+CETS among the most expensive software point in Table 8 [9]. Compatibility remains the other barrier: Song et al. report that SoftBound+CETS raises false alarms on many SPEC benchmarks because integer-pointer casts and uninstrumented libraries break metadata propagation [4].

WatchdogLite reaches 29–45% overhead for the same model by adding new hardware instructions [24]; PTSan reaches comparable overhead with, at most, commodity address masking. PTSan keeps the identity-based check but moves the metadata into the pointer value and a flat table, so propagation costs nothing, no shadow stack or shadow memory exists to desynchronize, and checks remain optimizable IR on commodity hardware.

Per-Object Tracking with Encoded or Tagged Pointers. Low-Fat Pointers derive bounds from the pointer value itself by segregating the heap and stack into size-class regions, at low runtime cost and near-native memory [25, 26]. The encoding is the guarantee's limit: bounds are the allocation size class, not the object, so overflows into rounding padding pass undetected [12], there is no temporal protection, and the assumption that out-of-bounds pointers do not escape breaks real programs [4]. Delta Pointers fold an overflow check into spare pointer bits at very low cost, but by design detect only overflows, with no underflow or temporal coverage [27]. PTSan also keeps pointers machine-sized and pays a higher runtime cost than these systems, but its tag indexes exact per-object bounds, so padding offers no refuge and the same identifier supports temporal checking.

Hardware-assisted tagging compares small colors instead of bounds: HWASan and Arm MTE tag pointers and memory and trap on a mismatch [28, 29], with tags of 4 to 8 bits making detection probabilistic at collision odds as high as one in sixteen. The hardware only carries the tag cheaply in the pointer; HWASan still loads and compares the memory tag in software at every access, which RSan measures at 132–268% overhead [8]. Only Arm MTE checks the tag in hardware, and that needs MTE-capable silicon. PTSan's 16 bits are an index, not a color: a check compares the access range against exact bounds, so spatial enforcement is not probabilistic at all, and only temporal protection degrades with identifier reuse.

CUP is the closest prior design. Like PTSan, it uses a pointer-carried identifier to index exact per-object bounds, and it supports more simultaneously active identifiers [10]. A CUP enriched pointer replaces the address with a capability identifier and a 32-bit offset. When the identifier is reused, the stale offset is applied to the new object's base, so the pointer no longer retains the old address needed to distinguish the two allocations. Thus the temporal protection is trivially and deterministically evadable: for a base pointer, `malloc(A); free(A); malloc(B); use(A)` is not caught. PTSan instead preserves both the canonical address and the identifier; identifier reuse hides a stale access only when the replacement object's bounds also cover the old address. This representation also explains the performance difference: CUP must reload the base and reconstruct each enriched address, whereas PTSan's metadata lookup remains a separable check that LLVM can optimize and Intel LAM can accelerate. CUP consequently incurs roughly three times PTSan's runtime overhead.

Garbage-Collected Capability C. Fil-C gives each pointer an invisible capability and uses a concurrent collector so that `free` deterministically disables all pointers to the freed object [30, 31, 32]. This is the strongest temporal story available for C, at multiples of native runtime [31], and it requires a garbage-collected runtime, a new ABI, and rebuilding all linked code [33]. PTSan occupies the recompile-only point: weaker temporal protection and an uninstrumented-library boundary, in exchange for sanitizer-style deployment at a fraction of the cost.

10. Future Work

PTSan's compact pointer representation establishes a practical point in the design space, and its optimization pipeline creates a foundation for exploring richer representations. We are investigating larger authority spaces and new identifier-management schemes that support more simultaneously live allocations without weakening PTSan's spatial and temporal protections.

A complementary direction builds on PTSan's optimizer-visible checks through proof-guided and proof-verified loop specialization. In this approach, profiling identifies hot loops, and candidate guards express the data-structure invariants under which selected checks are redundant. CodeHawk-C, our abstract-interpretation framework [34], generates the corresponding memory-safety proof obligations and verifies that the guards discharge every obligation associated with an elided check. Runtime dispatch selects the specialized loop only when its verified guard holds and otherwise executes the fully checked version, preserving PTSan's enforcement while allowing application invariants to remove substantially more hot-path work.

We are also extending PTSan's response to detected violations. PTSan already blocks an invalid access before it executes; that enforcement point can also serve as a controlled transfer to an appropriate recovery boundary, such as request cancellation, input rejection, an existing error path, or worker restart. Our initial work uses a violation's call-graph path to locate developer-written error handling and route control to it automatically. The longer-term goal is policy-guided recovery that preserves availability without turning recovery into silent continuation: every violation remains reported, the offending operation remains blocked, and recovery proceeds only through an isolation or error-

handling path whose restoration of program invariants can itself be verified.

11. Conclusion

We presented PTSan, an LLVM sanitizer that makes pointer-based memory safety practical by carrying a compact object identifier in the pointer itself and keeping bounds and lifetime state in small fixed tables. Because identity travels with the pointer value, ordinary dataflow propagates it for free, and the work that remains at an access stays ordinary LLVM IR that the compiler can hoist, merge, and elide. On SPEC CPU 2017, PTSan adds 57.2% runtime overhead on stock x86-64 hardware, 54.7% on ARM64, and 46.4% with Intel LAM, with 1.014× memory use, while detecting inter-object and temporal violations in MSET that location-based sanitizers miss. These results show that the long-standing cost gap between location-based and identity-based sanitizers is not fundamental: when identity propagation is free and checks are left visible to the optimizer, the gap closes to parity, with the residual cost of tag stripping reduced in software by ID-strip hoisting and removed entirely by hardware address masking.

Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under the Enhanced SBOM for Optimized Software Sustainment (E-BOSS) program, contract HR001124C0486. The views, opinions, and findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] Cybersecurity and Infrastructure Security Agency. *The Urgent Need for Memory Safety in Software Products*. Sept. 20, 2023. URL: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products (visited on 05/04/2026).

[2] Chromium Project. *Memory Safety*. URL: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 05/04/2026).

[3] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013.

[4] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. "SoK: Sanitizing for Security". In: *2019 IEEE Symposium on Security and Privacy*. 2019.

[5] Richard Fang, Rohan Bindu, Akul Gupta, and Daniel Kang. "LLM Agents Can Autonomously Exploit One-day Vulnerabilities". In: *CoRR* abs/2404.08144 (2024). arXiv: 2404.08144.

[6] Richard Fang, Rohan Bindu, Akul Gupta, Qiusi Zhan, and Daniel Kang. "Teams of LLM Agents Can Exploit Zero-Day Vulnerabilities". In: *CoRR* abs/2406.01637 (2024). arXiv: 2406.01637.

[7] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *2012 USENIX Annual Technical Conference*. 2012.

[8] Floris Gorter and Cristiano Giuffrida. "RangeSanitizer: Detecting Memory Errors with Efficient Range Checks". In: *Proceedings of the 34th USENIX Security Symposium*. 2025.

[9] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. "SoftBound+CETS Revisited: More Than a Decade Later". In: *EuroSec '24*. 2024.

[10] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. "CUP: Comprehensive User-Space Protection for C/C++". In: *ASIACCS '18*. 2018.

[11] Intel Corporation. *Linear Address Masking (LAM), Core Ultra Processor Series 3 Datasheet, Volume 1 of 2*. Intel Explore Design Center. URL: https://edc.intel.com/content/www/us/en/design/platforms/core-processor-series-3-datasheet-volume-1-of-2/linear-address-masking-lam/ (visited on 07/17/2026).

[12] Emanuel Q. Vintila, Philipp Zieris, and Julian Horsch. "Evaluating the Effectiveness of Memory Safety Sanitizers". In: *2025 IEEE Symposium on Security and Privacy*. 2025.

[13] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In: *PLDI*. 2009.

[14] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. "CETS: Compiler-Enforced Temporal Safety for C". In: *ISMM '10*. 2010.

[15] National Vulnerability Database. *CVE-2025-57807 Detail*. CVE-2025-57807. URL: https://nvd.nist.gov/vuln/detail/CVE-2025-57807 (visited on 05/11/2026).

[16] The Linux Kernel Developers. *Complete Virtual Memory Map (x86-64)*. Linux kernel documentation. URL: https://docs.kernel.org/arch/x86/x86_64/mm.html (visited on 07/17/2026).

[17] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016.

[18] Misiker Tadesse Aga and Todd Austin. "Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization". In: *CGO '19*. 2019.

[19] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent ByungHoon Kang. "SaVioR: Thwarting Stack-Based Memory Safety Violations by Randomizing Stack Layout". In: *IEEE Transactions on Dependable and Secure Computing* 19.4 (2022).

[20] E. A. Dinic. “Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation”. In: *Soviet Mathematics Doklady* 11 (1970). English translation of Doklady Akademii Nauk SSSR 194(4):754–757.

[21] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. “Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation”. In: *2024 IEEE Symposium on Security and Privacy*. 2024.

[22] LLVM Project. *LLVM test-suite Guide*. URL: https://llvm.org/docs/TestSuiteGuide.html (visited on 06/09/2026).

[23] Michael Larabel and Matthew Tippett. *Phoronix Test Suite*. Open-source automated benchmarking software. URL: https://www.phoronix-test-suite.com/ (visited on 06/11/2026).

[24] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. “WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking”. In: *CGO ’14*. 2014.

[25] Gregory J. Duck and Roland H. C. Yap. “Heap Bounds Protection with Low Fat Pointers”. In: *CC 2016*. 2016.

[26] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. “Stack Bounds Protection with Low Fat Pointers”. In: *NDSS Symposium 2017*. 2017.

[27] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. “Delta Pointers: Buffer Overflow Checks Without the Checks”. In: *EuroSys ’18*. 2018.

[28] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. *Memory Tagging and how it improves C/C++ memory safety*. 2018. arXiv: 1802.09517.

[29] Arm Ltd. *Armv8.5-A Memory Tagging Extension*. White paper. 2019. URL: https://developer.arm.com/documentation/102925/latest/ (visited on 06/10/2026).

[30] Fil-C Project. *Fil-C*. URL: https://fil-c.org/ (visited on 05/05/2026).

[31] Fil-C Project. *InvisiCaps: The Fil-C Capability Model*. URL: https://fil-c.org/invisicaps (visited on 05/05/2026).

[32] Fil-C Project. *How Fil-C Works*. URL: https://fil-c.org/how (visited on 05/05/2026).

[33] Fil-C Project. *Fil-C Runtime*. URL: https://fil-c.org/runtime (visited on 05/05/2026).

[34] Aarno Labs. *CodeHawk Analyzer*. May 4, 2026. URL: https://github.com/static-analysis-engineering/CodeHawk (visited on 05/04/2026).

Appendix A.
Additional Evaluation Data

This appendix collects per-benchmark and supplementary data for the evaluation of Section 8 and the related work of Section 9.

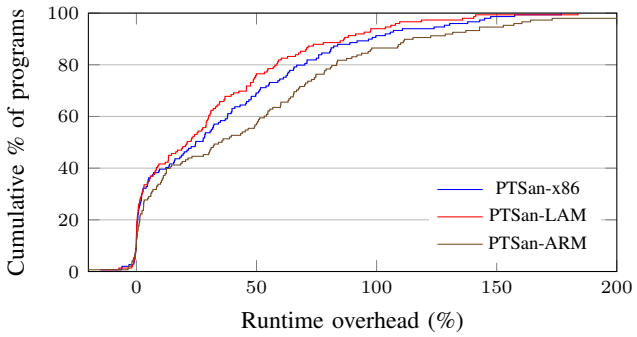


Figure 4. Distribution of PTSan runtime overhead over the measured LLVM MultiSource programs (149 on x86-64, 148 on ARM64). The axis is clipped at 200%; two ARM64 measurements exceed it (`sqlite3` at 253% and `lambda` at 534%).

Figure 4 shows the distribution of per-program overhead on the LLVM MultiSource benchmarks summarized in Table 3.

TABLE 6. PER-BENCHMARK PEAK PHYSICAL MEMORY OVERHEAD ON SPEC CPU 2017 (CGROUP V2 `memory.peak`).

Benchmark	Native MiB	ASan	RSan	PTSan
SPECrate 2017 Integer				
500.perlbench_r	209	4.44×	4.54×	1.000×
505.mcf_r	616	1.47×	1.57×	1.002×
520.omnetpp_r	262	4.45×	4.80×	0.989×
523.xalancbmk_r	543	5.29×	4.63×	1.001×
525.x264_r	1727	failed	1.31×	1.015×
531.deepsjeng_r	707	1.00×	1.04×	1.005×
541.leela_r	31	34.32×	37.18×	1.121×
557.xz_r	783	failed	1.45×	1.003×
SPECrate 2017 Floating Point				
508.namd_r	167	2.96×	2.89×	1.014×
510.parest_r	418	failed	4.36×	1.007×
519.lbm_r	417	1.13×	1.07×	1.005×
538.imagick_r	291	2.75×	2.76×	1.010×
544.nab_r	152	3.50×	3.91×	1.016×
Geomean		**3.33**×	**3.01**×	**1.014**×
Max		34.32×	37.18×	1.121×
Completed		10/13	13/13	13/13

TABLE 7. MSET DETECTION RESULTS FOR CASES WITH SATISFIED PRECONDITIONS.

System	Linear	Non-linear	Type conf.	UAF/UAR	Double-free	Misuse-free
ASan	46/66	0/54	6/18	8/16	4/4	20/20
RSan	40/72	0/54	10/24	2/8	4/4	0/20
PTSan	54/72	36/54	12/18	16/16	4/4	20/20

Table 6 reports the per-benchmark peak-memory measurements behind the aggregates of Section 8.3. For memory, all systems are measured against the same native baseline on the same machine; the mixed-language benchmarks and `502.gcc_r` are excluded because RSan does not cover or build them (Section 8.1), and "failed" marks ASan runs that crashed before completion. Table 7 restricts the local MSET runs of Section 8.4 to supported cases only, excluding precondition failures from both numerator and denominator. Denominators differ per system because each sanitizer perturbs allocation behavior differently, so different tests fail their preconditions under each; SoftBound+CETS is omitted because only the full-denominator convention is published.

Table 8 summarizes guarantees, costs, and deployment models across the location-based and identity-based sanitizers discussed in Sections 8 and 9.

Table 9 reports the per-benchmark data behind the ablation summary of Section 8.5.

Table 10 isolates the runtime cost of stack temporal protection with and without LAM.

A.1. Server Workloads

Table 11 gives the full results behind Section 8.2.3. We ran each Phoronix profile in its default "target" configuration in three configurations (native, PTSan-x86, and PTSan-LAM). The workloads are:

- **Redis** (SET, 50 connections): `redis-benchmark` issues SET commands of short string values against the single-threaded `redis-server` over 50 connections.
- **PostgreSQL** (pgbench, buffer test, 1 thread, read-write): the TPC-B-like mixed SELECT/UPDATE/INSERT profile with one client thread, committing to the write-ahead log on each transaction.
- **NGINX** (concurrency 1): `wrk` fetches a static page over HTTPS at one concurrent connection, exercising the TLS handshake and request-parsing path, with static delivery through `sendfile`.
- **OpenSSL** (RSA-4096): `openssl speed rsa4096`, reporting sign and verify operations per second; asymmetric crypto dominated by bignum modular exponentiation.
- **memcached** (Set:Get 1:100): `memtier_benchmark` drives the in-memory key/value store at a 1:100 Set:Get ratio, exercising the slab allocator and hash table.
- **Apache** (100 concurrency): `ab` serves a static page at 100 concurrent requests under the prefork worker model.
- **SQLite** (insert, single thread): replays 2,500 autocommit inserts three times against an on-disk WAL database from one thread, timing wall-clock seconds; every commit syncs, making it storage-bound.

TABLE 8. DESIGN-SPACE SUMMARY OF REPRESENTATIVE SANITIZERS: SPEC GEOMEAN OVERHEADS AND MSET DETECTION COVERAGE.

System	Temporal	Runtime	Memory	MSET inter-object	MSET temporal	Deployment model
Location-based						
ASan [7]	quarantine	117%[a]	3.33×[a]	90/126	32/40	recompile
RSan [8]	quarantine	51%[b]	3.01×[b]	79/126	14/40	recompile + allocator/loader
Low-Fat [25, 26]	none	54%[c]	1.0–1.1×[c]	—	—	recompile + allocator
HWASan [28]	probabilistic	132–268%[c]	1.04–1.06×[c]	—	—	recompile (Arm TBI)
Identity-based						
SoftBound+CETS [13, 14]	deterministic	140–161%[c]	2.7×[c]	126/126	40/40[c]	recompile, per-pointer metadata
CUP [10]	probabilistic	158%[c]	n/r	—	—	recompile + allocator
Fil-C [31]	deterministic	up to ∼300%[c]	n/r	—	—	new ABI + GC runtime
PTSan	probabilistic	**57%**/55%/46%[d]	**1.014**×	**126/126**	**40/40**	recompile (LAM: custom kernel)

Memory is peak resident growth over native; MSET columns are the inter-object spatial and summed temporal buckets (full denominators); — not available; n/r not reported. [a] Measured locally (Section 8.1); ASan runtime is from an earlier pass on the PTSan host. [b] Measured locally against its own baseline on a 13-benchmark subset. [c] As published: Low-Fat and CUP on SPEC CPU 2006; SoftBound+CETS as the modern port on the SPEC CPU 2017 C subset [9]; HWASan as measured on Arm TBI and Intel LAM hardware [8]; Fil-C as self-reported in its project documentation [31]. [d] x86-64 / ARM64 / x86-64 with LAM (Section 8.2).

TABLE 9. PER-BENCHMARK OPTIMIZATION ABLATION ON SPEC CPU 2017 (CUMULATIVE WATERFALL): COLUMN $-n$ IS THE CONFIGURATION AFTER THE FIRST n REMOVALS, IN THE ROW ORDER OF TABLE 5.

Benchmark	Full	−1	−2	−3	−4	−5	−6	−7	−8	−9	−10
SPECrate 2017 Integer											
500.perlbench_r	117.2%	181.4%	177.5%	181.4%	186.1%	185.9%	187.0%	197.8%	208.1%	202.9%	246.7%
502.gcc_r	61.4%	109.4%	113.8%	116.5%	114.5%	115.5%	121.7%	130.0%	134.0%	125.7%	136.0%
505.mcf_r	32.3%	36.2%	44.5%	42.9%	42.8%	41.8%	43.5%	53.4%	53.2%	54.0%	60.9%
520.omnetpp_r	61.8%	73.5%	74.9%	74.3%	83.5%	84.4%	83.5%	87.8%	89.2%	90.7%	89.1%
523.xalancbmk_r	57.7%	75.8%	74.0%	75.9%	80.3%	80.2%	80.8%	88.5%	88.1%	92.6%	110.4%
525.x264_r	40.6%	44.8%	56.1%	71.9%	79.2%	88.5%	88.4%	154.6%	159.3%	164.9%	208.7%
531.deepsjeng_r	49.8%	56.9%	62.3%	66.7%	79.9%	81.2%	81.5%	98.8%	103.2%	103.5%	130.9%
541.leela_r	57.3%	64.2%	71.4%	79.8%	110.7%	111.9%	111.5%	118.6%	121.9%	120.6%	136.1%
557.xz_r	31.9%	34.0%	37.4%	45.8%	47.3%	48.3%	47.2%	52.4%	55.4%	53.5%	68.8%
Geomean (INT)	**55.0%**	**70.6%**	**75.2%**	**80.2%**	**87.8%**	**89.3%**	**90.0%**	**104.6%**	**107.5%**	**107.3%**	**125.2%**
SPECrate 2017 Floating Point											
507.cactuBSSN_r	57.2%	57.2%	58.8%	60.0%	60.1%	76.0%	76.2%	182.4%	181.8%	237.5%	312.0%
508.namd_r	62.9%	73.9%	80.0%	101.1%	99.1%	116.9%	119.4%	203.3%	194.4%	162.0%	210.4%
510.parest_r	39.3%	39.0%	58.5%	60.4%	59.4%	110.5%	112.9%	130.0%	130.3%	167.0%	191.4%
511.povray_r	118.3%	136.5%	138.6%	148.6%	458.6%	458.6%	459.8%	491.8%	499.5%	491.4%	523.1%
519.lbm_r	-3.1%	-2.1%	-2.9%	-9.8%	-9.6%	-9.4%	-11.3%	33.4%	34.6%	32.5%	88.7%
526.blender_r	8.8%	18.0%	31.7%	34.1%	31.9%	34.6%	34.8%	54.1%	50.6%	56.6%	72.8%
538.imagick_r	36.7%	45.4%	45.8%	45.6%	75.8%	75.7%	76.3%	78.6%	84.2%	90.2%	93.2%
544.nab_r	12.4%	17.9%	22.3%	23.8%	23.8%	26.9%	27.5%	-8.0%	-7.8%	-9.2%	32.8%
Geomean (FP)	**37.3%**	**43.3%**	**49.3%**	**51.7%**	**71.2%**	**82.3%**	**82.6%**	**111.1%**	**111.1%**	**117.5%**	**157.7%**
Geomean (combined)	**46.4%**	**57.2%**	**62.5%**	**66.2%**	**79.8%**	**86.0%**	**86.5%**	**107.6%**	**109.2%**	**112.0%**	**139.9%**

TABLE 10. PER-BENCHMARK RUNTIME OVERHEAD ON SPEC CPU 2017 WITH STACK TEMPORAL PROTECTION DISABLED AND ENABLED.

	SPECrate 2017 Integer			
	PTSan-LAM		PTSan-x86	
Benchmark	Disabled	Enabled	Disabled	Enabled
500.perlbench_r	117.1%	125.8%	179.4%	197.1%
502.gcc_r	61.5%	65.8%	112.2%	116.9%
505.mcf_r	32.5%	33.1%	36.5%	35.6%
520.omnetpp_r	62.5%	68.5%	72.4%	79.6%
523.xalancbmk_r	58.9%	63.3%	76.6%	85.9%
525.x264_r	40.5%	49.6%	44.6%	53.4%
531.deepsjeng_r	50.7%	56.9%	58.6%	66.5%
541.leela_r	56.8%	70.5%	63.9%	77.8%
557.xz_r	32.1%	32.2%	32.2%	34.4%
Geomean (INT)	**55.3%**	**61.0%**	**70.6%**	**77.8%**

	SPECrate 2017 Floating Point			
	PTSan-LAM		PTSan-x86	
Benchmark	Disabled	Enabled	Disabled	Enabled
507.cactuBSSN_r	58.1%	58.6%	60.0%	57.7%
508.namd_r	62.8%	65.2%	73.8%	74.5%
510.parest_r	39.2%	35.8%	38.8%	39.7%
511.povray_r	117.3%	218.3%	133.3%	240.1%
519.lbm_r	-0.9%	-1.6%	-2.1%	-2.1%
526.blender_r	8.8%	10.3%	17.8%	20.9%
538.imagick_r	36.1%	46.0%	44.7%	54.1%
544.nab_r	12.7%	-27.3%	18.0%	-21.2%
Geomean (FP)	**37.6%**	**37.9%**	**43.2%**	**44.3%**
Geomean (combined)	**46.7%**	**49.7%**	**57.1%**	**61.2%**

TABLE 11. PTSAN OVERHEAD AND PEAK LIVE-ID DEMAND ON PHORONIX TEST SUITE SERVER AND CRYPTOGRAPHIC WORKLOADS, EACH IN ITS DEFAULT CONFIGURATION ON THE INTEL MACHINE OF SECTION 8.1. OVERHEAD IS RELATIVE TO NATIVE PERFORMANCE; A NEGATIVE VALUE IS FASTER THAN NATIVE. SQLITE REPORTS WALL-CLOCK TIME, SO ITS OVERHEAD IS ADDED LATENCY.

Workload	Native	PTSan-x86	PTSan-LAM	Peak IDs
Redis	293,701 RPS	26.0%	25.0%	16,411
PostgreSQL	505 TPS	16.0%	12.0%	9,247
NGINX	1,802 RPS	44.0%	2.0%	7,577
OpenSSL (sign)	323 sign/s	9.0%	8.0%	7,505
OpenSSL (verify)	23,200 verify/s	3.0%	3.0%	7,505
memcached	524,000 Op/s	9.0%	8.0%	706
Apache	16,500 RPS	0.0%	0.0%	380
SQLite	2.5 s	5.0%	6.0%	296