PSan: Towards Hybrid Metadata Scheme for Efficient Pointer Checking

Shengjie Xu, Eric Liu, Wei Huang, Ilya Grishchenko, David Lie
University of Toronto
{shengjie.xu, ec.liu, wh.huang }@mail.utoronto.ca, {ilya.grishchenko, david.lie}@utoronto.ca

***Abstract*—Memory safety remains at risk for programs written in unsafe languages like C. Pointer-checking schemes provide memory safety protection by attaching metadata for each pointer and checking them before dereference. Previously, sanitizers maintaining large per-pointer metadata (e.g., pointer bounds) were stuck with shadow memory for metadata storage, which incurs high overhead. Although fat pointers (i.e., instrumenting programs to *inline* metadata with pointers) incur less overhead, they introduce incompatibility issues to the instrumented programs, and are thus not considered by software-only sanitizers yet.**

In this paper, we push the status quo on adopting fat pointers for software-only pointer checking schemes and evaluate the benefit of this approach. We present PSan (short for "Pointer Sanitizer"), the first memory safety sanitizer that enables both inline and shadow memory metadata *simultaneously* in the same program. To reduce the overhead from shadow memory, PSan uses whole-program analysis and transformation to inline the metadata whenever possible, while using shadow memory only when necessary for compatibility. PSan-instrumented programs preserve binary compatibility with third-party uninstrumented code. In addition, PSan's framework decouples metadata management from checking, facilitating its augmentation with additional checkers.

We evaluate the benefit of metadata inlining and observe that PSan's hybrid scheme reduces the runtime and memory overhead. Specifically, PSan incurs 40% lower overhead than popular memory checker SoftBoundCETS, which utilizes only shadow memory. Predictably, using inline metadata has a higher performance improvement when it can be applied to the majority of pointers in the program.

***Index Terms*—Memory Safety, Fat Pointer, Sanitizer.**

1. Introduction

Memory safety remains a challenging problem in C and C++, which lack safeguards against misuse, leading to vulnerabilities like buffer overflows and use-after-free that may cause crashes, data breaches, or remote code execution [1]. Prior work has shown that compiler instrumentation is an effective solution for memory safety vulnerabilities. Schemes can attach metadata to pointers (e.g., pointer bounds for spatial safety) and check the metadata before pointer dereferences to catch unsafe operations [2], [3], [4], [5], [6]. Compared to alternative approaches (e.g., AddressSanitizer [7] or Control Flow Integrity [8]), these pointer checking schemes have the strongest error detection capability because they utilize more information about allocated objects (e.g., size and lifetime) to implement stronger checks than alternatives. However, they incur high overhead on the instrumented programs, sometimes exceeding 300%. Therefore, a significant amount of research effort has been devoted to reducing the overhead of memory checking schemes while maintaining their error detection capability.

Besides the checks themselves, a major source of overhead is pointer metadata storage and lookup. Early works store metadata in *shadow memory*— logically, an array indexed by all valid virtual addresses. It preserves object memory layout at the cost of additional latency for metadata lookup. Subsequent proposals prefer inlining metadata with pointers (known as *fat pointers*) for better performance. However, this approach currently requires instrumentation of the entire program, breaking compatibility with third-party binaries. Consequently, finding an efficient yet compatible way to maintain pointer metadata remains elusive.

In this paper, we show our findings on incorporating fat pointers for pointer checking without breaking compatibility. We present PSan, a pointer-checking sanitizer that utilizes both shadow memory and fat pointers. PSan achieves this by *demand-driven metadata inlining*: it uses whole-program static analysis to identify pointers applicable for inline metadata and widens them into fat pointers, while remaining pointers that cannot be transformed due to compatibility constraints use shadow memory. This approach achieves the performance benefit of fat pointers while preserving the compatibility of shadow memory.

To facilitate code reuse for future research on pointer checking, we implemented PSan as a general framework that supports flexible extension by decoupling checking and metadata management. The framework encapsulates metadata storage, retrieval, and propagation. Checker components implement the security policy and optimizations. This allows us to implement spatial and temporal memory safety protection [2] with around 600 lines of code each. Both checks can be enabled simultaneously, providing stronger error detection.

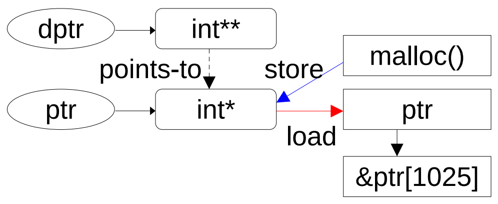
We evaluate PSan with programs from the Olden benchmark and SPEC 2017. Non-surprisingly, the effect of PSan utilization depends on the proportion of inlined pointers. Therefore, PSan's demand-driven metadata inlining effectively reduces the overhead for Olden benchmarks, where PSan can use inline metadata for almost all pointers in half of the programs. At the same time, due to the limitations in the employed pointer analysis, the PSan prototype cannot inline metadata for enough pointers in SPEC 2017 programs, resulting in marginal performance improvements. Nonetheless, PSan outperforms SoftBoundCETS [9] (a well-known shadow-memory-based pointer-checking solution) in our experiments. PSan's bug-finding ability is also tested on the Juliet test suite [10], a commonly used tool for evaluating runtime error detection of checking tools, and PSan catches all relevant vulnerabilities.

This paper makes the following contributions:

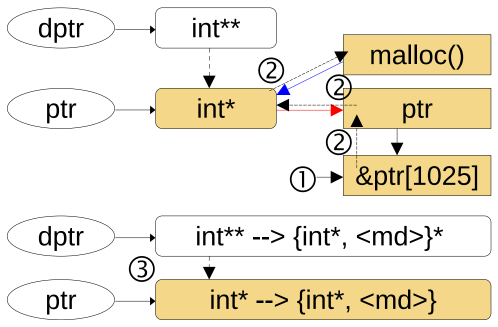
- We present the first pointer-checking scheme that simultaneously supports inline metadata (fat pointers) and shadow memory, preserving binary compatibility with external code. The scheme's demand-driven metadata inlining algorithm demonstrates performance overhead improvements for programs that permit metadata inlining.
- We implement our pointer-checking scheme as a prototype PSan—a software-only sanitizer that provides spatial and temporal memory safety.
- We evaluate PSan on popular benchmarks and demonstrate performance improvements when using PSan on programs that permit metadata inlining. In all our experiments, PSan outperforms the popular shadow-memory-only checker SoftBoundCETS, incurring on average 40% lower overhead.

```
1  int* ptr;
2  int** dptr = &ptr;
3  void foo(void) {
4    ptr = (int*) malloc(sizeof(int)*1025);
5    ptr[1025] = 0; /* out-of-bound */
6  }
7  // int* ptr2;
8  // void setptr(int v) {
9  //   dptr = (v>0? &ptr: &ptr2);
10 // }
```

(a) Code Example



(b) Graph Representation



(c) Algorithm Steps

Figure 1: MIFP's VAPG-based Type Transform

2. Background: Metadata Inlining using Points-to Graph

Points-to graph is a representation of pointer analysis results widely used by static analyzers [11]. MIFP [12] pioneers the use of a *Value-argumented* points-to graph (VAPG, "extended points-to graph" in the original paper) for widening pointers to fat pointers. PSan repurposes and improves upon this approach. MIFP [12] is designed to enhance CHERI's spatial safety protection. CHERI Concentrate [13] proposed fat pointer compression to reduce overhead, making the bounds inaccurate when object sizes exceed the platform-dependent threshold (e.g., 4KB for 64-bit systems). MIFP analyzes which pointers may carry inaccurate bounds, and attaches uncompressed bounds for vulnerable ones.

MIFP overview. A program shown in Figure 1a contains an off-by-one error at line 5. The vanilla CHERI compressed capability pointer will carry an approximated bound that misses this error. To guarantee the detection of the error, MIFP replaces the `int*` capability pointer type with a struct containing (1) the original capability pointer (still represented as `int*` in Figure 1), and (2) an additional uncompressed bounds as the extra metadata that can detect the off-by-one error. MIFP starts from line 5 where the unsafe dereference is made, finds the source of pointer value that would flow to the dereference site (the `malloc` call at line 4), and transforms all pointer types along the way to ensure that the accurate bounds can be passed from the object allocation to the pointer use site (where MIFP instruments an additional check using the accurate bounds). Because `malloc` returned pointer is stored in the global variable `ptr` at line 1, MIFP will transform `ptr`'s type to embed the accurate bounds. Because `dptr` at line 2 takes the address of `ptr`, MIFP also transforms the points-to type of `dptr` to show that it points to the widened fat pointer. The size of `dptr` itself is not changed because `dptr` is not used in an unsafe dereference. MIFP's algorithm automates all the instrumentation decisions above. The use of VAPG simplifies the handling of points-to relationships when transforming pointer types. We now walk through the algorithm in details.

Expanded MIFP walkthrough. Given the program in Figure 1a, MIFP first builds the VAPG as shown in Figure 1b. We hide unnecessary nodes for clarity. Nodes in rectangular boxes (right side) are *expression* nodes representing pointer values in the source program. There are three expression nodes in the example figure: the `malloc` call at Line 4, the access to `ptr` at Line 5, and the address computation of `&ptr[1025]` at Line 5 before the store. Nodes in ellipses (left side) are *allocation* nodes representing allocations or variable definitions. Nodes in rounded rectangles (middle) are *cell* nodes representing "alias sets" of objects, essentially an indistinguishable set of objects from the instrumentation perspective. If Lines 7-10 in Figure 1a are uncommented, `dptr` can point to either `ptr` or a new global variable `ptr2`, then `ptr2` will have a separate allocation node but share the same cell node `int*` because it is indistinguishable from `dptr`. MIFP uses cell nodes as the smallest element for widening. That is, if the algorithm widens a pointer, it widens all the other pointers sharing the same cell to keep a consistent static type (e.g., for `dptr` and its dependent objects). The edges among cell nodes represent type relationships (and thus dependencies) among them. For example, as `dptr` is initialized with `&ptr`, there is an `points-to` edge from the `int**` cell of `dptr` to the `int*` cell of `ptr`. This edge means that the type of cell `int**` is updated when the cell `int*` is updated. Edges from or to expression nodes represent dataflows (e.g., loads and stores). If a pointer expression should carry metadata, these edges form the dataflow path the metadata should propagate along with the pointer value. VAPG extends existing Points-to Graph [11] with dataflow edges, and it treats functions as in-memory objects.

After VAPG is built, MIFP runs the type transform algorithm as shown in Figure 1c. Circled numbers label the steps when the algorithm uses the nodes and edges. Step 1 identifies pointers requiring additional metadata using static analysis, which finds pointers that may carry inaccurate bounds and may be dereferenced in code that is not statically safe. We highlight such pointers in orange. In the example above, the expression `&ptr[1025]` created from line 5 is marked because it is dereferenced out-of-bound, and the compressed bounds will miss the error.

Next, in Step 2, the algorithm performs a backward analysis to identify all pointer expressions and cells that require widening. Starting from each node found in the previous step (`&ptr[1025]`), the algorithm keeps traversing back along incoming edges and marks all visited nodes for widening, including expression `ptr`, cell `int*`, and the `malloc` expression.

In Step 3, MIFP derives a new type for each node in the subgraph with only cell nodes and edges between them, as it contains all type dependencies. In particular, the algorithm assigns new types until the fixedpoint is reached. First, pointers requiring metadata (e.g., the `int*` cell) are replaced with structs containing the original pointers and the additional metadata. Then, whenever a type is converted, all other cells whose type depends on that type (e.g., the `int**` cell from `dptr`) are updated accordingly. The algorithm keeps updating the types until no further changes are needed. MIFP instruments the program to use the new types for processed allocations and expressions afterward.

3. Design of PSan

PSan is a pointer-checking sanitizer that uses inline metadata (fat pointers) and out-of-band metadata (shadow memory) simultaneously. PSan inlines metadata whenever this program transform is safe; otherwise, it stores metadata in the shadow memory. This approach allows PSan to benefit from the lower overhead of inline metadata without compromising generality or compatibility. PSan encapsulates security enforcement logic in checker components, which are responsible for identifying pointers requiring metadata and instrumenting checks. The framework handles most metadata maintenance, making individual security checkers more straightforward to implement and extend.

PSan improves over MIFP by addressing the following key challenges:

Supporting type-unsafe operations PSan tolerates arbitrary pointer casts and unions in programs. MIFP cannot support them because they create inconsistent views on data types in memory, making subsequent type transforms unsound. PSan solves this problem by protecting type-unsafe pointers with shadow memory, whereas MIFP can only inline metadata.

Protecting pointers exposed to external code PSan uses shadow memory to store metadata for pointers that cannot have type transforms (e.g., reachable from external call argument). MIFP cannot protect such pointers.

Handling code constructs impairing pointer analysis Constructing the VAPG depends on an inter-procedural pointer analysis. This analysis does not support several implementation-specific code constructs (e.g., pointer vectors). When encountering them, it may create incorrect points-to graphs, resulting in broken programs. PSan uses an allowlist-based heuristic to accept safe code constructs and identify unsupported ones, then prunes reachable code from unsupported constructs to prevent them from confusing the pointer analysis.

To overcome these challenges, PSan employs a disciplined approach shown in Figure 2. PSan first partitions the input program into an analyzable slice (blue) and an unanalyzable slice (red). We use the term *unanalyzable* to denote code regions that contain constructs preventing our analysis from safely inlining pointer metadata. PSan uses a specialized taint analysis to identify pointers and objects that may be affected by or reachable from such unanalyzable code, and applies the inline scheme only to unaffected pointers using the algorithm adapted from MIFP. The metadata for the remaining pointers are maintained in the shadow memory.

One challenge is that the taint analysis alone cannot distinguish benign and bad pointer casts. Consider three programs with pointer casts, where `T` and `U` are two incompatible types. Program A contains a bad cast of a pointer `T*` to `U*` directly. Program B contains a bad cast chain across function boundaries. It casts pointer `T*` to `void*` in function `f1`, passes it to another function `f2`, and casts it to `U*` there. Program C casts `T*` to `void*` in `f1`, passes it to `f2`, and `f2` performs a benign cast back to `T*`. While the taint analysis can taint the bad cast in Program A, it cannot taint the bad cast in Program B while leaving the benign cast in Program C untainted, as doing so would

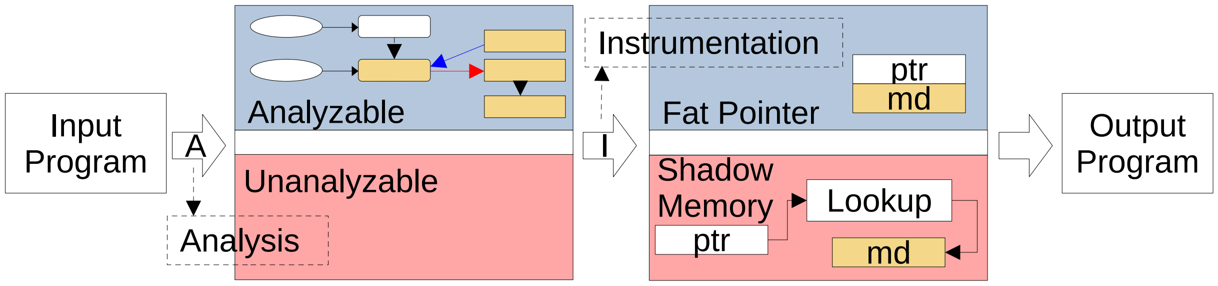


Figure 2: PSan's Demand-Driven Metadata Inlining

require knowing the real type of the pointed object, which requires inter-procedural pointer analysis. To support benign casting in Program C, PSan adopts the separation of duty with (1) the initial tainting of only value-independent bad casts in Program A, and (2) the construction algorithm for the VAPG detecting value-dependent bad casts in Program B.

Because inlining the pointer metadata is equivalent to transforming pointers into structs that include the metadata in addition to the original pointers, the rest of this paper uses *type transform* to describe the transform needed to inline pointer metadata. This also includes passing metadata across function calls and returns because they are equivalent to transforming the function prototype to include the metadata.

Algorithm overview. Figure 3 shows the overview of PSan. Nodes represent Program IR, data, and other intermediate states. Algorithm steps are labeled with circled numbers. Steps 1-5 correspond to the analysis stage in Figure2, and Step 6 - to the instrumentation stage. PSan requires whole-program IR as input; any functions or global variables defined outside the input IR are treated as external code and PSan prohibits type transform on data exposed to them.

Step 1: Pre-transform. Compiler optimizations before the generation of the whole-program IR can introduce unnecessary pointer casts and destroy type information. For instance, assuming there is a global variable `T* ptr` pointing to struct type `T`, the code in line 1 below can be optimized into line 2 by LLVM's InstCombine pass:

```
1   ptr = (T*) malloc(sizeof(T)); // good
2 *(void*)&ptr=malloc(sizeof(T)); // bad
```

This transform prevents PSan from discovering the real type of `*ptr` and disables metadata inlining for pointers in `T`. To reduce unnecessary pointer casts, PSan starts with pre-transforming the programs and trying to remove such casts. For pointers that have lost type information (becoming `void*`), we run use-based type inference to find a consistent type for the pointed objects. This step makes more code analyzable and improves the coverage of the inline metadata scheme. In fact, without this step, PSan would be unable to inline any metadata in evaluated programs. Besides casts, our implementation detects and inlines allocation wrappers (e.g., `xmalloc`) so that the intra-procedural analysis during graph construction can look for the correct types inside the callers of the allocation wrappers (i.e., callers of `xmalloc`). Using this information, even if the `malloc` calls in the example above are replaced with `xmalloc` (which uses only `void*`), PSan still finds the object type `T` for the result pointer.

Step 2: Initial tainting and pruning. To support general programs, PSan needs to (1) remove code constructs that mislead or interfere with pointer analysis and VAPG construction (Step 3 below), and (2) reduce analysis time spent on pointers and objects not applicable to metadata inlining. Therefore, the second step is to run a static taint analysis to identify the unanalyzable slice of code and then prune it for the subsequent analysis. The taint analysis finds all pointer expressions (program IR instructions, function arguments, and addresses of global variables) that may be used or produced by unanalyzable code constructs. To avoid using heavyweight pointer analysis during static tainting, the tainting in this step is *imperfect* in that (1) it does not track taints across memory loads and stores, and (2) it assumes there are no value-dependent bad casts. Step 3 (discussed below) handles these cases by invoking pointer analysis to propagate taints.

The exact definition of unanalyzable constructs is implementation-specific. We consider code unanalyzable if it contains (1) arbitrary pointer casts, (2) calls to externally defined functions, or (3) other code constructs that are hard for an implementation to analyze or transform (e.g., vectors). However, our prototype has support for calls to functions like `memcpy` and `memset`, so such calls are considered analyzable. In the implementation, we populate an "allowlist" with known-supported code constructs, and anything outside the list is considered unanalyzable.

PSan then takes unanalyzable code features as sources for the static taint analysis and partitions the program. Once the algorithm finds an unanalyzable operation on a pointer, all its points-to objects are excluded from the analyzable slice because they may be accessed from the unanalyzable slice. In addition, if the code stores a pointer to a memory location in the unanalyzable slice, we treat it as unanalyzable and exclude its points-to objects from the analyzable slice[1]. By removing all pointer expressions reachable from unanalyzable code, PSan ensures that all the memory accesses to pointers in the analyzable slice can be identified and

1. These rules are implemented in both Steps 2 and 3.

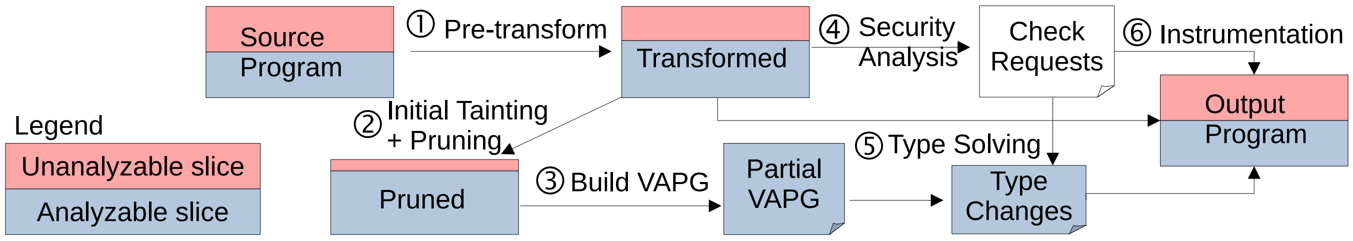


Figure 3: PSan Algorithm Overview

transformed.

After the taint analysis, the algorithm creates a pruned version of the program suitable for the subsequent graph-based analysis. This version contains the entire analyzable slice and certain instructions (like function calls) from the unanalyzable slice necessary for analysis. If a tainted value has users in the pruned IR (e.g., storing a tainted value into untainted memory), PSan replaces the value with *placeholders* (implemented as calls to externally-defined functions) and redirects the users to use the placeholder values. This replacement effectively mitigates the impact of unanalyzable values on the subsequent pointer analysis. The pruning also removes all integer or floating-point instructions that have no impact on pointer analysis. In our evaluation, the pruning removes >40% of instructions on most programs, and it reduces the pointer analysis time by 40%-99%.

Step 3: Graph construction. The third step is to run the pointer analysis and construct the VAPG. We extended the MIFP algorithm by introducing taints for graph nodes to detect dataflow-dependent type mismatches and propagate taints across memory loads and stores. During the graph construction, if we find that there is an inconsistent type (e.g., casting a `T*` pointer to `void*` in one function and casting it back to `U*` instead of `T*` in another function), we taint the affected objects and pointers, and they will be excluded from metadata inlining. This step finalizes PSan's static tainting and program slicing for analyzable versus unanalyzable code.

Step 4: Security analysis. This step invokes the checker components to decide which pointers require checking (thus requiring metadata) and where the checks should be placed. In our prototype, the spatial and temporal safety checker is applied to the pointer operands in all memory load/stores and `memcpy`/`memset`/`memmove` calls.

Step 5: Type solving. After determining which pointers require metadata for checking, Step 5 of the algorithm identifies new types for the involved pointers and objects in the VAPG. Pointers carrying metadata will be marked for transform to structs (fat pointers) that have extra space for metadata, and all other types depending on them (functions, objects containing pointers, and pointers to them) are updated accordingly.

Step 6: Instrumentation. In the last step, PSan instruments the program using the transform information from the previous step. PSan updates all code involved in type changes, instruments code for metadata initialization and maintenance, and invokes checker components to instrument checks using the metadata.

4. Implementation

In this section, we describe our implementation of PSan based on LLVM 15 with typed pointers. We use the SVF framework [14] for pointer analysis.

PSan is an LLVM IR instrumentation tool that expects whole-program LLVM IR as an input and produces the corresponding instrumented IR. It supports the instrumentation of C programs. The compilation pipeline using PSan consists of (1) collecting the whole-program IR (e.g., via gllvm [15]), (2) running PSan to instrument the IR, and finally (3) compiling the instrumented IR with optimizations to get executables. The prototype assumes that functions and global variables defined in the whole-program IR are inaccessible from external code unless (1) they are special symbols (e.g., `main`), or (2) they are address-taken.

To support programs modifying pointers in uninstrumented libc functions (e.g., calling `qsort` to sort a pointer array), we implemented the workaround introduced in Intel MPX [16]: each pointer metadata stored in the shadow memory makes a copy of the pointer value so that if the uninstrumented code modifies the pointer, PSan-instrumented code loading the metadata can detect the mismatch between the backup value and current value and then ignore the stale metadata. Our experiments show that this workaround increases the total run time by less than 5% for most programs.

Code size. PSan consists of a generic framework (17 KLOC excluding SVF), individual checkers for specific memory safety properties (1.1 KLOC combined), and a shadow memory implementation (520 LOC). The complexity of the framework mainly comes from the VAPG-based algorithm (5.2 KLOC) and instrumentation (5.0 KLOC). Of our currently implemented checkers, the spatial safety checker has ~550 LOC, and the temporal safety checker has ~530 LOC.

4.1. PSan as a Pointer-checking Framework

Figure 4 shows the design of the PSan framework. We abbreviate “metadata” as MD. Boxes with colored backgrounds represent different instrumentation stages (initialization - blue, propagation - yellow, check insertion - red, and miscellaneous - white). The implementation encapsulates all checking-specific logic in the checker components

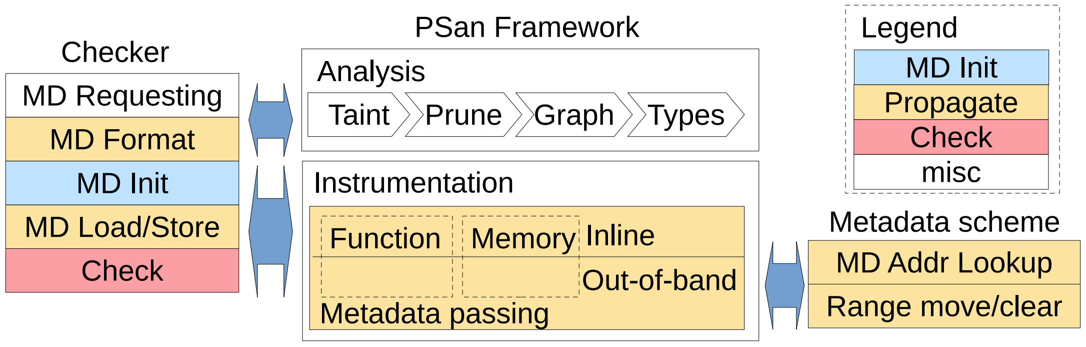


Figure 4: PSan Framework Design

and splits the implementation of the shadow memory into the metadata scheme component. Each component consists of (1) a corresponding C++ class with virtual functions and (2) (optional) a runtime library written in C. To reduce the performance impact of the runtime library, our implementation links the LLVM IR of the runtime libraries during instrumentation. This way, we can use existing compiler optimizations after PSan's instrumentation to optimize the entire program.

Checkers. PSan allows arbitrary checkers to be implemented. A checker needs to specify at least (1) where to instrument checks ("MD Requesting" in Figure 4), (2) a metadata representation ("MD Format"), (3) initial metadata values ("MD Init"), and (4) how to instrument runtime checks ("Check"). The checker can implement custom metadata compression by distinguishing metadata representation in memory and in registers ("MD Format") and overriding the default metadata load/store ("MD Load/Store") instrumentation. The checker is also responsible for optimizations on the checks (e.g., elimination, coalescing, or hoisting). The framework handles the rest of the analysis and instrumentation, transparently utilizing both metadata schemes and supporting any combination of checkers.

Metadata scheme. During the metadata passing, PSan uses the inline scheme for function prototypes and object allocations in the VAPG (thus in the analyzable slice) and uses the shadow memory otherwise. The shadow memory implementation is also decoupled from the framework. We currently implement a two-level hierarchical lookup table used in prior works [16], [17].

4.2. Checker Implementation

To provide memory safety protection for programs and facilitate evaluation for PSan in Section 5, our implementation includes a spatial memory safety checker and a temporal memory safety checker (Table 1). Both checkers insert checks before every memory load and store and `memset`/`memcpy`/`memmove` call, and employ intra-procedural analyses to eliminate redundant or statically-safe checks.

Spatial safety checker. The spatial safety checker implements traditional bounds checking. The metadata is an

TABLE 1: Spatial and Temporal Checker Specifications in PSan

Spatial Checker

Metadata	`void* lb, void* ub`

MD Init	`lb = &obj`
	`ub = lb + sizeof(obj)`

Check	`!(addr<lb || addr+size>ub`
	`|| addr>ULONG_MAX-size)`

Temporal Checker

Metadata	`uint64_t* lockaddr,`
	`uint64_t key`

MD Init	`lockaddr = allocate_lock(),`
	`*lockaddr = key =`
	`init_key(count++, dynamic)`
	Stack allocations share frame locks and keys

Check	`*lockaddr == key`

Check free()	`*lockaddr == key &&`
	is_dynamic(`key`)
	deallocate_lock(lockaddr) after checking

Finalization	Deallocate frame locks before function returns

address pair for object bounds: a lower bound `lb` and an upper bound `ub`. When an object is allocated, the lower bound is set to the base address of the object (`&obj`), and the upper bound is set to be one past the last byte of the object (`lb + sizeof(obj)`). When the bounds for a constant pointer expression are requested (e.g., `&a[42]` where `a` is a global array), the checker looks through the expression, finds the underlying object, and sets the corresponding bounds. If the metadata is not empty, a bounds check will fail if the memory access range goes below (`addr < lb`) or above (`addr + size > ub`) the permitted range, or if the size overflows. We implemented the traditional dominance-based redundant check elimination, along with check coalescing for struct member access and for "adjacent" checks (i.e., checks on the same pointer within the same basic block).

Temporal safety checker. The temporal safety checker implements the traditional lock-and-key scheme [18] to detect use-after-free and double-free errors.

For each object with a non-static lifetime (i.e., they can be deallocated at some time during execution), the checker associates a unique lock value (a 64-bit integer) with the object. We derive the lock value from a global counter to ensure its uniqueness. During the execution, the checker examines the pointer metadata, including a pointer to the lock (`lockaddr`) and the lock value (`key`). A mismatch between the key and the lock value indicates a temporal safety violation.

To allocate and deallocate the locks, the checker uses `allocate_lock` and `deallocate_lock` functions, respectively. We manage locks in a free list separate from application data (as in SoftBoundCETS [18]) to prevent confusing application data with locks when a use-after-free is triggered. The lock deallocation will destroy the lock value. All stack-allocated objects in the same function share the lock and key; we refer to this shared lock as a frame lock. For each function with a frame lock, the checker will instrument code to deallocate the frame lock before each return.

If a pointer has metadata (`lockaddr` is not `NULL`), a temporal safety check loads the lock value and compares it with the key; the check fails if it does not match the lock value. Besides checking dereferences, the temporal safety checker also performs checks before `free` calls to catch double-free errors and deallocate the object lock. This check ensures that the key is from a dynamic allocation (`is_dynamic(key)`, encoded in the key value) in addition to the lock-and-key comparison to prevent double-freeing of a stack-allocated object.

5. Evaluation

In this section, we evaluate PSan on (1) its ability to provide memory safety protection, and (2) how well it reduces the runtime and memory overhead of pointer checking. To run a program with PSan instrumentation, we compile it with `O1` optimization flag (auto-vectorizations disabled), collect the whole-program IR (e.g., using gllvm [15]), run PSan to produce the instrumented IR, and finally compile the IR with `O3` flag to produce executables.

Porting SoftBoundCETS. We compare PSan's security and performance with SoftBoundCETS [9], one of the most widely used pointer checkers that provide a level of security comparable to that of PSan. SoftBoundCETS features spatial and temporal safety checking using only shadow memory. To evaluate its implementation along with PSan, we split its instrumentation pass and runtime library from SoftBoundCETS [9]'s LLVM 12 fork and ported it to LLVM 15. We disable shadow memory pre-allocation (making allocations on-demand) because it halts the program in initialization in our setup. For a fair comparison, we mitigate differences in instrumentation and optimizations by dropping SoftBoundCETS' changes to the LLVM optimizations, and we built the runtime library into an LLVM bitcode file, which can be linked into the instrumented IR for better optimizations.

5.1. Security Evaluation

To evaluate PSan's memory safety protection strength, we ran it on the Juliet 1.3 C/C++ test suite [10] and compared the number of passed tests with the reported numbers from SoftBoundCETS. Each test case contains a "good" (bug-free) version and a "bad" (vulnerable) version of the program. We compile and instrument both versions and check the execution results. We consider a test case passes if (1) the good version executes correctly, and (2) the bad version aborts with error messages from PSan checkers.

Out of 64,099 test cases in the Juliet test suite, we select 11,252 C test cases from categories corresponding to spatial safety and temporal safety. From these test cases, we excluded 5,862 and used 5,390 test cases for the evaluation.

Excluded test cases. We excluded 798 spatial and temporal safety cases based on their categories. CWE123 (write-what-where condition) is omitted because the test cases deliberately let I/O functions (from console or network) overwrite pointers. PSan permits this behavior to support programs that depend on it. We omit CWE680 (integer overflow to buffer overflow) because they could require a gigantic `malloc` on LP64, which freezes the program and halts the testing. We also omit CWE761 (free inside buffer) because the PSan prototype is not implemented to detect this error yet. Because the prototype uses decoupled spatial and temporal checkers, the temporal checker cannot use pointer bounds to identify the start address of the freed buffer. Thus, it cannot validate the pointer argument to `free` alone. To support the detection of this error, PSan can be extended to pass metadata from the spatial checker to the temporal one.

Table 2 lists the reasons for the rest of the excluded cases (5,064 in total). We categorize excluded test cases into the following three types according to the reasons.

The first kind ("Opt") is for test cases whose invalid memory accesses get optimized away by the compiler. Because the current PSan prototype expects optimized input, we cannot disable the optimization to preserve these invalid accesses. Therefore, they did not reach the IR input for PSan instrumentation, and we cannot trigger them in the output executable. For example, most bad versions have all the code optimized away, leaving only print-related calls. Some spatial safety test cases contain one smaller buffer and one larger buffer, and the bad version overflows the smaller one by copying data from the larger one. The compiler can optimize away the smaller buffer completely by "merging" the smaller buffer into the larger one. The optimized version transforms accesses to the smaller buffer into in-bounds accesses into the larger buffer.

"Func" category is for test cases whose memory errors are triggered in library functions we did not instrument (e.g., `str(n)cat`/`str(n)cpy` in libc and print-related functions). To support detection of such errors, one can extend PSan with manually written wrappers for these library functions that also request checks on pointer arguments, as done in SoftBoundCETS. Because SoftBoundCETS contains wrappers for `str(n)cat`/`str(n)cpy`, it can find

TABLE 2: Security Evaluation on Juliet Test Suite

Category	Total	Excluded			Tested	(% Passed)
		Opt[1]	Func[2]	Misc[3]		
CWE121_Stack_Based_Buffer_Overflow	4036	1514	886	72	1564	(100%)
CWE122_Heap_Based_Buffer_Overflow	2504	494	570	224	1216	(100%)
CWE124_Buffer_Underwrite	1288	0	424	0	864	(100%)
CWE126_Buffer_Overread	972	24	18	90	840	(100%)
CWE127_Buffer_Underread	1288	146	424	0	718	(100%)
CWE415_Double_Free	228	120	0	0	108	(100%)
CWE416_Use_After_Free	138	0	58	0	80	(100%)
Sum	10454	2298	2380	386	5390	

[1] Invalid memory accesses are optimized away by the compiler.
[2] Error is triggered in uninstrumented library functions (e.g., strcpy).
[3] No invalid memory accesses or they are indistinguishable from valid ones in IR.

the invalid pointer arguments before calling into actual implementations.

"Misc" test cases are either free from invalid accesses or contain overflows within a single object. Some test cases that are free from invalid accesses (presumably targeting static code analyzers) use `sizeof` on pointers to allocate objects instead of allocated type (i.e., using `sizeof(ptr)` instead of `sizeof(*ptr)` as the size argument for `malloc`), but they use allocated types with the same size as pointers in 64-bit environment (`int64_t` or `double`), creating no errors at runtime. The remaining "Misc" test cases contain *intra-object* overflow from the first field (e.g., `memcpy(ptr->buffer, src, sizeof(*ptr))` where `buffer` is the first element in the struct pointed by `ptr`). In this case, `ptr` and `ptr->buffer` point to the same address in IR. Supporting these test cases require (1) extending PSan's spatial safety checker with *subobject bounds checking*, i.e. narrow the bounds when driving a pointer to a struct member to just cover the intended member, and (2) modify LLVM optimizations to preserve struct member address computation (LLVM `GetElementPointer` instructions).

Test case setup and modifications. For test cases only triggering errors with specific inputs from `stdin`, we specify the input data in the test script to ensure that the bad version can trigger the error at run time. To run test cases that require generated random numbers to fall in specific ranges, we modify the support library (where macros for random number generations are defined) so that the program can read provided values from environment variables instead of generating random ones. For test cases taking inputs from socket connections, we implement the corresponding TCP server and client to pass the data. We manually modify test cases running TCP servers (i.e., test cases with "listen_socket" in names) to add `setsockopt` call with `SO_REUSEADDR` to avoid socket `bind` failures.

Result. For the considered 5,390 test cases, as shown in Table 2, PSan correctly detects all errors in the bad versions and reports no false positives in the good versions. SoftBound demonstrated identical performance to PSan. We conclude that the PSan prototype accurately catches all memory safety errors within its design scope, offering security guarantees equivalent to those of SoftBoundCETS.

5.2. Performance Evaluation

For performance evaluation, we choose six out of eight C programs from SPEC2017 and all ten programs from the Olden benchmark. We select Olden because it is used in the evaluation of MIFP [12], on which PSan is based. We exclude `perlbench` and `gcc` from SPEC2017 because the PSan Prototype currently does not produce fully functional executables for them. We run all programs on Ubuntu 25.04 (Linux 6.14 kernel) with an Intel i7-7700 processor and 16GB of RAM.

We investigate the following research questions:

RQ1: Overall Performance: What is the overhead of PSan? How well does it compare to using only shadow memory in PSan and SoftBoundCETS?

RQ2: Overhead Decomposition: How to interpret the results from RQ1? It is further divided into:

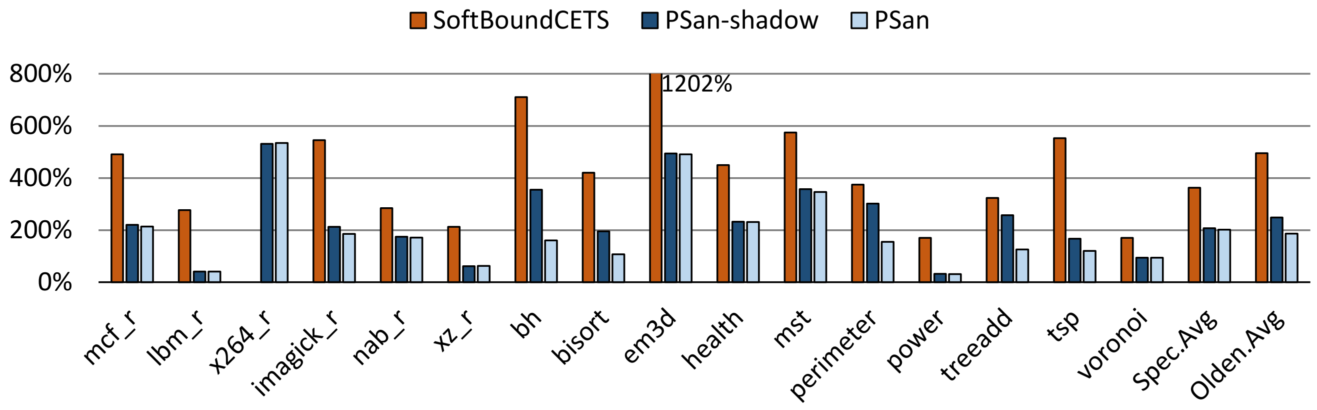
a) How well does PSan's metadata inlining algorithm perform in reducing metadata overhead?
b) How much of the overhead comes from checking and how much from metadata? More overhead from metadata indicates more room for overhead reduction from PSan's metadata inlining.

We now present the answers to the questions above in separate subsections below.

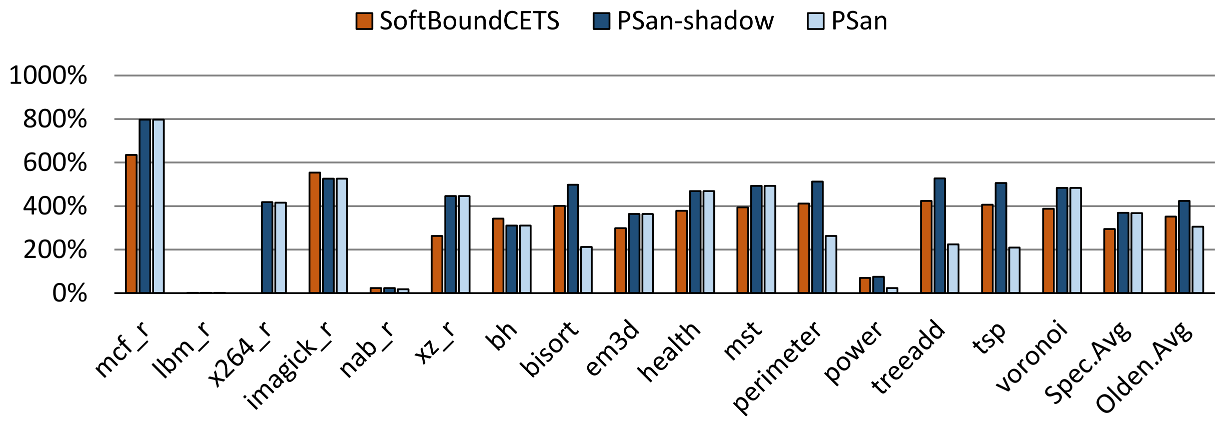
5.2.1. RQ1: Overall performance.

To compare the overhead for enabling or disabling pointer metadata inlining, we implemented an additional alternate version of PSan that we refer to as PSan-shadow, which uses only shadow memory. For the compilation workflow, we first create a single whole-program LLVM IR for each benchmark program with O1 optimization but with vectorization disabled[2] as the shared input. We then build the baseline version with O3 optimization from this IR. For PSan, we instrument the O1-optimized IR, then run Clang with O3 compilation on the

2. PSan prototype cannot propagate metadata for pointer vectors, thus enabling vectorization on the input IR will weaken PSan's protection.



(a) Runtime Overhead



(b) Memory Overhead

Figure 5: Performance Evaluation Results

instrumented IR. For SoftBoundCETS, we immediately run O3 optimization on the input IR, then invoke the instrumentation, followed by O3 compilation. We enable both spatial and temporal safety in PSan and SoftBoundCETS.

TABLE 3: Average Overhead for Each Tool

	SPEC 2017		Olden	
	Perf	Mem	Perf	Mem
SoftBoundCETS [9]	362%	295%	495%	351%
PSan-shadow	207%	369%	249%	424%
PSan	201%	367%	186%	305%

Figure 5a and 5b show the runtime and memory overhead of each version on individual benchmarks. The average overhead is summarized in Table 3. Because the ported SoftBoundCETS fails to recover after a false positive in `x264_r`, we omit the data for this program when computing averages.

Compared to PSan-shadow, PSan's inlined metadata organizations reduced the runtime overhead by 3% and 25% and memory overhead by <1% and 28% for SPEC2017 and Olden, respectively.

Compared with SoftBoundCETS, PSan runs 53% faster while using 18% more memory in SPEC 2017. The higher memory usage comes from the extra copy of the pointer value for detecting stale metadata. The performance improvements are significantly larger in Older programs, where PSan can inline more pointer metadata. Specifically, PSan is 108% faster and uses 10% less memory.

5.2.2. RQ2: Overhead Decomposition. To study the contribution of overhead from checking and metadata, we implement and run two additional modes of PSan: (1) a `stat` mode that collects runtime statistics during execution, and (2) a `nocheck` version without runtime checks but with the propagations and register pressure from metadata[3]. Figure 6 plots the collected statistics and the checking overhead contribution.

3. `nocheck` mode replaces checks with assembly code comments that simply list the name of registers storing metadata.

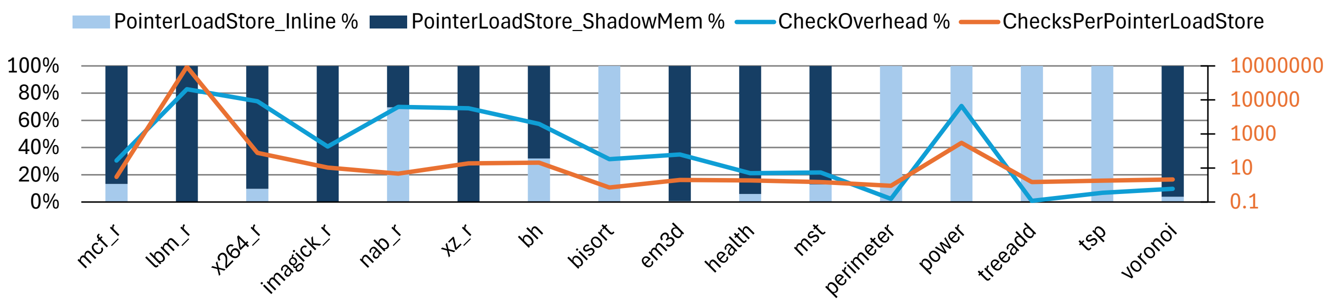


Figure 6: (Dynamic) Inline Metadata Coverage and Checking Overhead

Inline metadata coverage. The bar chart in Figure 6 plots the percentage ratio of pointer loads and stores using inline (light blue, "PointerLoadStore_Inline") or out-of-band metadata (dark blue, "PointerLoadStore_ShadowMem") reported from `stat` mode runs. This ratio indicates the rate of inlining. The inline metadata has almost full coverage in five Olden programs. Programs with high inline metadata coverage can achieve more overhead reduction because fat pointers preserve the locality between a pointer and its metadata, eliminating metadata address lookup. The program `bh` has significant overhead reduction despite only 32% inline metadata load/store, because it has 7.7×10^7 `memcpy`/`memset` calls on analyzable objects. Such calls would require accompanying metadata copying or clear operations if accessed objects use shadow memory, thus resulting in increased overhead without the inline scheme. However, the overhead reduction is not linear with the coverage of inline metadata; the benefit may not be clear until the coverage is very high, depending on the processor. For example, `544.nab_r` has about 70% pointer load/stores use inline metadata, which reduces the data cache misses and the instruction count. Still, the overhead reduction is only about 2% due to the doubled instruction TLB misses.

Checking vs metadata overhead. To compute the breakdown of overhead on checking and metadata, we compare the run time of the `nocheck` mode against normal PSan's operation: the overhead of the `nocheck` version is solely due to metadata operations, and the difference is the overhead from checking. The blue line in Figure 6 ("CheckOverhead") plots the computed contribution in percentage. Most Olden programs have <40% overhead from checking (except `bh` and `power`), which means that metadata operations contribute to the bulk of overhead. Therefore, Olden programs with high inline metadata coverage (`perimeter`, `treeadd`, and `tsp` in particular) get significant overhead reduction (28%-51%). In contrast, SPEC programs either have more overhead due to checking or lower inline metadata coverage. Therefore, compared to the PSan-shadow version, PSan's overhead reduction is still noticeable but lower.

Check-to-access ratio. Besides the overhead contribution computed from runtime difference, another metric that can verify our estimation on checking overhead is the ratio between the total number of checks and the number of pointer loads and stores. This approximates how often pointer's metadata is used (checked) after it is accessed in memory on average. Generally speaking, this ratio should be positively correlated with checking overhead contribution: more checks per pointer access means that the metadata load/store contributes less and checking contributes more to the overhead. We plot this ratio in orange lines using a logarithmic scale in Figure 6 ("ChecksPerPointerLoadStore"). As shown in the figure, the ratio generally follows the trend of the blue line (checking overhead). 10 out of 16 programs have less than 10 checks per pointer load/store, 4 out of 16 have 10-100 checks per pointer load/store, and the remaining two (`lbm_r` and `power`) have >100 checks per pointer load/store. `lbm_r` has 1.8×10^4 pointer load/stores in total but $1.6 \times 10^1 1$ checks. This result correlates with our evaluation of the overhead reduction resulting from metadata handling.

6. Discussion and Limitations

Analysis time reduction from pruning. PSan's program slicing and code pruning require heavyweight pointer analysis and VAPG construction. To reduce PSan's runtime, we restrict PSan's application to only the analyzable program slice, rather than the entire program. Most programs require less than half of the time on the inter-procedural pointer analysis stage with the pruning. For instance, it reduces the analysis time on `538.imagick_r` from about 400s to 23s (47s if including instrumentation) while still achieving a 13% overhead reduction in the end.

Evaluation on real-world applications. We tested PSan on real-world applications, including Pure-FTPd (FTP server), nginx (HTTP server), SQLite (Database), FFmpeg (Media processing), and larger applications such as Firefox (Browser). Pure-FTPd and nginx were stable with PSan instrumentations. However, for these applications, we did not observe measurable overhead attributable to PSan because both applications are I/O-bound. The rest of the applications do not currently work with PSan's instrumentation due to assertion failures, triggered when PSan detects errors in its instrumentation due to some of the limitations below.

Limitations due to pointer analysis. Although PSan's design tolerates code constructs that can impair pointer analysis (Section 3, discussion for Step 2), it still assumes that the rest of the pointer analysis results are accurate. However, we observed inaccurate pointer analysis results (missing pointer value flow relationships) in our experiments on real-world applications (e.g., in SQLite, FFmpeg), which caused terminations of instrumented programs. Improving SVF's interprocedural pointer analysis is required to support more complex programs.

Another consequence of the imprecision of the pointer analysis is that PSan must be conservative when encountering indirect calls. We currently prohibit type transforms on points-to objects of pointer arguments and return values for indirect calls and possible callees, which reduces the coverage of metadata inlining and increases overhead.

Limitations due to implementation effort. A number of limitations can be overcome with more implementation effort. First, our prototype does not yet support external shared libraries referencing symbols defined within the whole-program IR (e.g., plugins) because it assumes that no external code can reference these symbols unless the symbols are address-taken. Second, the prototype cannot detect invalid pointer arguments passed to external code (e.g., use an out-of-bound pointer in `strcpy`). SoftBoundCETS detects such issues by implementing wrappers that check pointer arguments before calling the real function. PSan's implementation would also require a mechanism (either in the runtime library or in the instrumentation code) to support checking these function calls on a case-by-case basis.

Inherent limitations. PSan shares several limitations with prior works [12], [17]. First, In general, it is difficult to definitively infer object sizes. PSan uses MIFP's heuristics for inferring object sizes, which relies on calls to `malloc` and `memcpy`, and it cannot update `sizeof` results in code not covered by the heuristics above. For example, if the code has a wrapper function for `memcpy` and prepares size arguments outside the wrapper, PSan will be unable to update these size arguments, and the `memcpy` may not copy enough bytes, resulting in program misbehavior. Second, PSan's instrumentation does not enforce the atomicity of metadata accesses. Therefore, to use multi-threaded code with concurrent access to pointers with PSan, such code must be ported to place such accesses in atomic regions.

Future work. There are a few directions that can increase the amount of metadata inlining and improve PSan's performance. For example, PSan does not yet support metadata inlining for code using C++ features like polymorphism. Adding such support requires redesigning VAPG's type system (currently aligned with LLVM) to capture C++ polymorphism. In addition, type-agnostic algorithms using `void*` (e.g., a generic linked list) currently prevent the metadata inlining as well. One way of tackling this problem is to duplicate the code to create a specialization for the data type.

7. Related Work

Memory safety has been a long-standing problem, with several surveys studying various enforcement designs [19], [20], [21], [2], including pointer checking and alternative approaches. Prior works have explored pointer checking in software [9], [17], [18] and hardware [16], [13], [5], [6], [4]. In this section, we discuss recent research directions for memory safety protection.

Tagged pointers. Several enforcement schemes use small metadata that fit into the unused high bits of pointer values. The metadata is referred to as pointer *tags*. The hard limit on metadata size reduces their protection granularity but improves performance. Low-Fat Pointer [22] manipulates object placements so the object bounds can be inferred from the high address bits. FRAMER [23], In-Fat Pointer [24], HeapCheck [25], and CECSan [26] use pointer tags to locate in-memory metadata. Besides storing an index for metadata lookup, PACMem [27] protects the pointer integrity with ARM Pointer Authentication. SPP [28] uses tagged pointers to protect applications using persistent memory (PM).

Memory tagging. Memory tagging schemes [29] attach metadata to each memory location instead of each pointer. This makes them tolerant to the loss of information about pointers and objects and results in a lower number of false positives. AddressSanitizer [7] is a popular sanitizer that uses memory tags to label the allocation state of each object so that it can catch accesses to deallocated or uninitialized memory. ARM MTE [30] introduces memory tagging into commodity hardware that can be used to detect spatial and temporal memory errors [31]. DMTI [32] uses ARM MTE with dynamic binary instrumentation to detect memory errors in C/C++ binaries. IntegriTag [33] repurposes Intel TME-MK for memory tagging to implement memory safety policies.

Optimizations. WPBound [34] hoists spatial safety checks outside loops. MemSafe [35] uses an inter-procedural pointer data flow graph to optimize temporal safety checks. Catamaran [36] offloads spatial safety checks to another thread to improve performance. There are also efforts to consolidate and remove redundant checks using static analysis [37] or combining static and dynamic analysis [38], [39].

Language extensions. Checked C [40] extends C with *dependent types* and annotations that enable expressing pointer bounds using program expressions, eliminating the need for extra metadata. 3C [41] automates porting existing code into Checked C.

8. Conclusion

In this paper, we present PSan, the first pointer-checking scheme that supports both pointer inlining and shadow memory. We demonstrate how PSan utilizes taint analysis to identify objects whose pointer metadata cannot be inlined, allowing inlining to be applied to the remaining objects. Our evaluation shows that PSan's approach—compared to the shadow-memory-only approach SoftBoundCETS—reduces runtime and memory overhead for memory safety enforcement. We believe PSan's demand-driven metadata inlining is an important step for lower-overhead pointer checking.

Acknowledgments

We thank the anonymous reviewers and shepherd for their insightful comments and helpful feedback, which improved this paper substantially. This research was supported in part by NSERC CRD Grant CRDPJ 541942-19, NSERC Discovery Grant RGPIN-2018-059, and a Research Contract with Huawei Canada. David Lie is support by Tier 1 Canada Research Chair CRC-2019-00242. Eric Liu was supported by an Ontario Graduate Scholarship.

References

[1] Office of the National Cyber Director, "Back to the building blocks: A path toward secure and measurable software," White House, Tech. Rep., 2024.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Porceedings of the 34th IEEE Symposium on Security and Privacy*, ser. Oakland '13, San Francisco, CA, USA, May 2013, pp. 48–62.

[3] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture*, ser. ISCA '14, Minneapolis, MN, USA, June 2014, pp. 457–468.

[4] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the C programming language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, Seattle, WA, USA, March 2008, p. 103–114. [Online]. Available: https://doi.org/10.1145/1346281.1346295

[5] S. Das, R. H. Unnithan, A. Menon, C. Rebeiro, and K. Veezhinathan, "SHAKTI-MS: A RISC-V processor for memory safety in C," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2019, Phoenix, AZ, USA, June 2019, p. 19–32. [Online]. Available: https://doi.org/10.1145/3316482.3326356

[6] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "WatchdogLite: Hardware-accelerated compiler-based pointer checking," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, Orlando, FL, USA, Feburary 2014, p. 175–184. [Online]. Available: https://doi.org/10.1145/2544137.2544147

[7] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[8] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3054924

[9] B. Orthen, O. Braunsdorf, P. Zieris, and J. Horsch, "Softbound+cets revisited: More than a decade later," in *Proceedings of the 17th European Workshop on Systems Security*, ser. EuroSec '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 22–28. [Online]. Available: https://doi.org/10.1145/3642974.3652285

[10] NIST, "Juliet test suite for C/C++," 2017. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[11] W. Wang, C. Barrett, and T. Wies, "Partitioned memory models for program analysis," in *Verification, Model Checking, and Abstract Interpretation*, A. Bouajjani and D. Monniaux, Eds. Cham: Springer International Publishing, 2017, pp. 539–558.

[12] S. Xu, E. Liu, W. Huang, and D. Lie, "MIFP: Selective fat-pointer bounds compression for accurate bounds checking," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 609–622. [Online]. Available: https://doi.org/10.1145/3607199.3607212

[13] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "Cheri concentrate: Practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, 2019.

[14] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[15] SRI International's Computer Science Laboratory, "Whole program llvm in go." [Online]. Available: https://github.com/SRI-CSL/gllvm

[16] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the Intel MPX system stack," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, June 2018. [Online]. Available: https://doi.org/10.1145/3224423

[17] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland, June 2009, p. 245–258. [Online]. Available: https://doi.org/10.1145/1542476.1542504

[18] ——, "CETS: compiler enforced temporal safety for C," *SIGPLAN Not.*, vol. 45, no. 8, p. 31–40, jun 2010. [Online]. Available: https://doi.org/10.1145/1837855.1806657

[19] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1275–1295.

[20] E. Vintila, P. Zieris, and J. Horsch, " Evaluating the Effectiveness of Memory Safety Sanitizers ," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 88–88. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00088

[21] M. Brohet and F. Regazzoni, "A survey on thwarting memory corruption in RISC-V," *ACM Comput. Surv.*, vol. 56, no. 2, sep 2023. [Online]. Available: https://doi.org/10.1145/3604906

[22] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ser. CCS '13, Berlin, Germany, November 2013, p. 721–732. [Online]. Available: https://doi.org/10.1145/2508859.2516713

[23] M. J. Nam, P. Akritidis, and D. J. Greaves, "FRAMER: A tagged-pointer capability system with memory safety applications," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, San Juan, Puerto Rico, USA, December 2019, p. 612–626. [Online]. Available: https://doi.org/10.1145/3359789.3359799

[24] S. Xu, W. Huang, and D. Lie, "In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 224–240. [Online]. Available: https://doi.org/10.1145/3445814.3446761

[25] G. Saileshwar, R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu, "Heapcheck: Low-cost hardware support for memory safety," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, jan 2022. [Online]. Available: https://doi.org/10.1145/3495152

[26] X. Wang, B. Zhang, C. Tang, and L. Zhang, "Highly comprehensive and efficient memory safety enforcement with pointer tagging," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2024, pp. 74–81.

[27] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1901–1915. [Online]. Available: https://doi.org/10.1145/3548606.3560598

[28] D. Stavrakakis, A. Panfil, M. Nam, and P. Bhatotia, "Spp: Safe persistent pointers for memory safety," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 37–52.

[29] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves c/c++ memory safety," 2018.

[30] *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*, Arm Limited, 2019, https://developer.arm.com/documentation/ddi0487/ea. Accessed 2023-06-30.

[31] M. Unterguggenberger, D. Schrammel, P. Nasahl, R. Schilling, L. Lamster, and S. Mangard, "Multi-tag: A hardware-software co-design for memory safety based on multi-granular memory tagging," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 177–189. [Online]. Available: https://doi.org/10.1145/3579856.3590331

[32] A. Hager-Clukas and K. Hohentanner, "Dmti: Accelerating memory error detection in precompiled c/c++ binaries with arm memory tagging extension," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1173–1185. [Online]. Available: https://doi.org/10.1145/3634737.3637655

[33] D. Schrammel, M. Unterguggenberger, L. Lamster, S. Sultana, K. Grewal, M. LeMay, D. M. Durham, and S. Mangard, "Memory tagging using cryptographic integrity on commodity x86 cpus," in *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, 2024, pp. 311–326.

[34] Y. Sui, D. Ye, Y. Su, and J. Xue, "Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1682–1699, 2016.

[35] M. S. Simpson and R. K. Barua, "Memsafe: Ensuring the spatial and temporal memory safety of c at runtime," in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 199–208.

[36] Y. Zhang, T. Liu, Z. Sun, Z. Chen, X. Li, and Z. Zuo, "Catamaran: Low-overhead memory safety enforcement via parallel acceleration," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 816–828. [Online]. Available: https://doi.org/10.1145/3597926.3598098

[37] T. Jung, F. Ritter, and S. Hack, "Pico: A presburger in-bounds check optimization for compiler-based memory safety instrumentations," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3460434

[38] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani, "SIMBER: Eliminating redundant memory bound checks via statistical inference," in *ICT Systems Security and Privacy Protection*, S. De Capitani di Vimercati and F. Martinelli, Eds. Cham: Springer International Publishing, 2017, pp. 413–426.

[39] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 479–494. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/zhang

[40] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked c: Making c safe by extension," in *2018 IEEE Cybersecurity Development (SecDev)*, 2018, pp. 53–60.

[41] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. Hicks, "C to checked c by 3c," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527322