Memory Safety Instrumentations in Practice: Usability, Performance, and Security Guarantees

Tina Jung
Saarland University
Saarland Informatics Campus (SIC)
Saarbrücken, Saarland, Germany
t.jung@cs.uni-saarland.de

Fabian Ritter
Saarland University
Saarland Informatics Campus (SIC)
Saarbrücken, Saarland, Germany
fabian.ritter@cs.uni-saarland.de

Sebastian Hack
Saarland University
Saarland Informatics Campus (SIC)
Saarbrücken, Saarland, Germany
hack@cs.uni-saarland.de

Abstract

Memory safety violations due to C's undefined behavior, although well researched, still cause security breaches year by year. The most dangerous reported violations are spatial safety violations, where objects are accessed outside of their bounds. A wide variety of spatial safety sanitizers promise easy usage, broad security guarantees, and a low execution time overhead. However, only few of them are actually used.

Instead of proposing yet another sanitizer, we dig deep into Low-Fat Pointers and SoftBound, two approaches to generate fast-to-execute safe programs with strong safety guarantees, and identify pain points in their usage. We found that seemingly small simplifying assumptions or limitations of the approaches often lead to spurious error reports.

On top of analyzing usability issues, we set up a framework that abstracts common tasks of memory safety instrumentations, such as finding locations for checks and eliminating redundant checks. This abstraction allows us to draw a fair comparison between approaches when it comes to execution time and the number of safe accesses. We use this framework to give novel insights into how many accesses are provably safe, and where to attribute execution time overhead.

Our findings help future research on memory safety instrumentations by identifying issues that current approaches face in their practical application. We make our LLVM-based instrumentation framework available to reduce the effort required to implement new instrumentations and to ease comparisons to Low-Fat Pointers and SoftBound.

CCS Concepts: • **Software and its engineering → Software safety**; **Compilers**; *Software testing and debugging*; *Software reliability*; • **Security and privacy → Software and application security**.

Keywords: Memory safety, C language, LLVM, SoftBound, Low-Fat Pointers

ACM Reference Format:
Tina Jung, Fabian Ritter, and Sebastian Hack. 2024. Memory Safety Instrumentations in Practice: Usability, Performance, and Security Guarantees. In *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO'25).* ACM, New York, NY, USA, 14 pages.

CGO'25, March 01–05, 2025, Las Vegas, NV, USA
2024.

1 Introduction

Memory safety is a concern of programming languages such as C, where arbitrary pointer arithmetic can be used to construct new pointers. The language permits only the creation of pointers to an object or one past its end, while out-of-bounds pointer arithmetic is undefined behavior. Accessing memory with out-of-bounds pointers is commonly referred to as a *spatial memory safety violation.*

Undefined behavior makes it easy to translate C programs to fast-to-execute machine code, because compilers only need to preserve the program's defined semantics and can ignore corner cases whose behavior is undefined. However, undefined behavior makes it hard for programmers to track down or even notice bugs, as the program might still behave as expected. From the security perspective, undefined behavior, and especially the resulting memory safety violations, pose a significant threat. In the CWE Top 25 ranking [8] of the most dangerous software vulnerabilities, out-of-bounds writes are the top of the list in 2023, and are part of the most stubborn weaknesses over the past five years [7].

Various tools have been proposed to guarantee the spatial memory safety of C programs [1, 3, 9, 10, 13, 16–18, 23, 26, 30–32]. However, as Song et al. [33] have shown in their survey on memory safety tools, it is very hard to reproduce the evaluations of these tools. Song et al. were able to verify 10 out of 37 sanitizer artifacts as functional. Regarding approaches with some level of spatial memory safety guarantees, only three out of 21 were executed successfully [10, 31, 32]. The reasons are manifold: Among the publicly available tools, some did not compile or failed to run on more than half of the SPEC CPU2006 benchmarks. The authors of tools that are not available did either not respond to requests for the source code, or had concerns about licensing or code quality.

Other surveys [20, 33, 36] compare memory safety approaches on paper or test the functionality of the artifacts, but they do not focus on an even comparison. E.g., Song et al. [33] used different compiler versions for the tools, for some tools benchmarks were patched while for others functions with false positives were excluded from sanitization.

In contrast to previous surveys, this paper provides a *detailed* comparison of two memory safety approaches, which are broadly applicable, give strong spatial safety guarantees, and introduce comparably low runtime overhead: Low-Fat Pointers [10] and SoftBound [23]. We implemented the approaches in MemInstrument[1], a novel instrumentation framework which serves as the basis for a fair comparison of memory safety approaches. To the best of our knowledge, this paper is the first to provide such a framework and thereby allow a fair comparison of the two approaches.

Although many compiler-based memory safety approaches seem different conceptually, the basic mechanism of how they place instrumentation code is very similar. Approaches that instrument pointer arithmetic or pointer dereferences [1, 5, 10, 17, 23], thereby tracking pointer values through the program, follow very basic common schemes. We exploit this observation in MemInstrument to abstract common tasks like determining the points for in-bounds checks and metadata updates. Additionally, common approach-independent check optimizations are shared between the approaches. This common infrastructure is the basis for a fair comparison, as it rules out sources of runtime differences which are not actually caused by the instrumentation, but rather due to differences in the compiler pipeline setup or check placement.

We evaluate the runtime performance of Low-Fat Pointers and SoftBound and investigate how optimizations and the compiler pipeline affect the runtime, as well as which parts of the approaches contribute how much to the runtime overhead. We found the approaches to be on par in terms of runtime, while one outperforms the other on individual benchmarks. The impact of the compiler pipeline insertion point is significant, when not carefully chosen and compared, one could conclude that either approach is faster by around 30%, everything else being unchanged.

Besides runtime performance, we discuss the spatial safety guarantees of the approaches in practice, i.e., how valid C programs are unexpectedly rejected and how memory safety violations remain unnoticed. We found that SoftBound struggles most with outdated or unavailable bounds metadata, causing spurious error reports or impairing the ability to detect errors. Low-Fat Pointers reports some usages of out-of-bounds pointers, which conflicts with programmer expectations as well as the LLVM IR specification, leading to error reports for out-of-bounds pointers which are not dereferenced. While those shortcomings impede ensuring memory safety for certain programs, we also show that most accesses in successfully instrumented programs are properly safety checked by Low-Fat Pointers and SoftBound.

In summary, the contributions of this paper are:

- A fair runtime comparison of the two memory safety approaches Low-Fat Pointers [10] and SoftBound [23].
- A novel evaluation of:
  - the implications of the limitations of both approaches in practice (Sections 4.6, 5.1.1 and 5.1.2).
  - insights which parts of the instrumentation contribute to the execution time overhead (Section 5.4).
  - the impact of simple optimizations and different compiler pipeline setups (Sections 5.3 and 5.5).
- A novel instrumentation framework to allow future research to easily implement new approaches and compare against already existing ones.

[1] https://github.com/cdl-saarland/MemInstrument/

2 Background

The C programming language has been around for 50 years now and a substantial amount of widely-used code is written in it. The fact that out-of-bounds accesses are undefined behavior in C allows C compilers to omit out-of-bounds checks in the generated code without proving accesses in-bounds through program analysis. This gives C a performance advantage over languages with mandated bounds checking like Java. However, this advantage comes at the cost of security as out-of-bounds accesses can result in severe security vulnerabilities. To counter these problems, several *memory safety instrumentation* approaches have been developed. They modify the compiler to insert check code during compilation and thus harden the compiled program against abuses of undefined behavior at the cost of a slower execution speed.

Memory safety instrumentations face three major challenges: giving ample safety guarantees, being compatible to uninstrumented libraries, and incurring low overhead in terms of runtime, binary size and memory usage. We discuss several classes of instrumentations in terms of these aspects to motivate why we chose Low-Fat Pointers [10] and SoftBound [23] for a more detailed comparison.

2.1 Existing Memory Safety Approaches

One category of approaches are *red-zone approaches*. Address Sanitizer [31], Valgrind's Memcheck [32] and Dr. Memory [3] belong to this class. Red-zones are memory areas around each allocation that are marked as inaccessible. These approaches detect errors when a red-zone rather than a valid object is accessed. Dr. Memory and Memcheck are binary instrumentation tools that detect a wide range of errors in addition to out-of-bounds accesses, such as memory leaks or use-after-frees. While they are highly compatible as no recompilation is required, Bruening and Zhao [3] report a mean runtime slowdown of 10.2× for Dr. Memory and a 20.4× for Memcheck on the SPEC CPU2006 benchmarks [14].

Address Sanitizer requires recompilation of the program, but the runtime slowdown averages at only 1.7× for SPEC CPU2006 compared to the programs optimized at `-O2`.

Red-zone-based approaches are inherently incomplete: They cannot detect pointers that go out of bounds but end up in another allocation.

Approaches that use *fat pointers* to pass along bounds metadata with the pointer [2, 16, 26] report an error when accesses end up in a different allocation. Fat pointers change

Table 1. Locations for instrumentation

Instrumentation Target	Task	SoftBound	Low-Fat Pointers

`... = load ptr`
`store x ptr`
	ensure safety	in-bounds check	in-bounds check

`@global_ptr`
`ptr = alloca sz`
`ptr = call malloc(sz)`
	record allocation	determine size	add section marker
			mirror, replace
			use custom `malloc`

`ptr = phi(ptr1 %b1, ptr2 %b2, ...)`
`ptr = select c ptr1 ptr2`
`ptr = gep src_ptr i1, ...`
	propagate through phi
	propagate through select
	determine `src_ptr`		

`ptr = load location`
`ptr = call f (...)`
`fun_arg_ptr`
	rely on invariant	load from trie
		load from shadow stack
		load from shadow stack
			assume in bounds

`store ptr location`
`return ptr`
`call f (ptr, ...)`
	establish invariant	store to trie
		store to shadow stack
		store to shadow stack
			in-bounds check

the encoding of pointers from plain pointer values to, e.g., a `struct` containing the pointer value and base and size of the object it should point to. This comes with the disadvantage of poor compatibility to uninstrumented libraries, as these libraries cannot handle the fat pointer encoding. Type casts also pose a problem as they can corrupt the propagated metadata. Due to such issues, these approaches might require the programmer to rewrite parts of their application. Austin et al. [2] report that their fat-pointer-based tool executes 2.3−6.4× more instructions on memory-intense benchmarks.

Learning from these drawbacks, other approaches [1, 5, 17, 23] store pointer bounds in a disjoint metadata space. Uninstrumented code then works as usual, and type casts do not break the code. However, some approaches [1, 5, 23] still require wrappers for uninstrumented functions. For example, if such a function returns a pointer, the metadata needs to be made available to ensure safety of further accesses. Nagarakatte et al. [23] achieve a performance overhead of 1.67× on selected SPEC benchmarks compared to an uninstrumented program. The approach of Jones and Kelly [17] suffers from a 12× slowdown in that evaluation [23]. Chen et al. [5] additionally cover temporal safety, segment confusion errors and memory leaks; they report a 13× slowdown on five SPEC CPU2017 [35] benchmarks.

Akritidis et al. [1] introduce *Baggy Bounds*, which stores size information of pointers in a bound table. It exploits allocation alignment and padding to recover the base pointer from the size and the pointer value itself. Building on this idea, Low-Fat Pointers [10] make the base and size information fully recoverable from the pointer value, without an additional datastructure. As part of their technique, allocations are aligned and padded, so that the resulting program is hardened against out-of-bounds accesses, but accesses to the padding will not be detected. Duck and Yap [10] report a mean slowdown of 1.56× on the SPEC CPU2006 benchmarks compared to the uninstrumented (`-O2`-optimized) program.

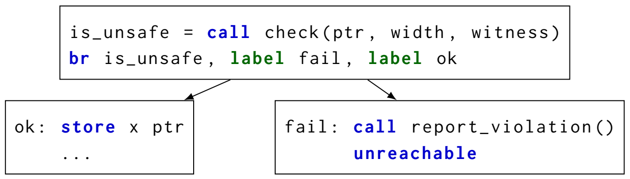
2.2 Low-Fat Pointers and SoftBound

Although not compared on a common ground until now, the published results indicate that SoftBound [23] and Low-Fat Pointers [10] are the most competitive approaches in terms of performance and spatial safety guarantees. Low-Fat Pointers give broad security guarantees and are (to the best of our knowledge) still the fastest self-reported approach to detect both under- and overflow errors (also stated, for example, by Kroes et al. [18]). SoftBound was designed with the learnings from fat pointer approaches [2, 16, 26], fares better in terms of runtime than other approaches with disjoint metadata [5, 17], and was often used to compare against before [4, 5, 28, 38], indicating its relevance in the research area.

We compare Low-Fat Pointers and SoftBound as they stand out with respect to their safety guarantees and runtime performance, while not being widely used. Additionally, the two approaches have less in common than they have with other approaches, and therefore promise (and deliver) more diverging reasons for their shortcomings.

3 Instrumentation Framework

Although the some aspects of memory safety instrumentations differ a lot, e.g., how they retrieve bound information for pointers, they share many tasks such as checking valid bounds when memory is accessed. Our instrumentation framework factors these commonalities out and determines code locations for metadata propagation and check placement – so-called *instrumentation targets* – with a shared strategy. Abstracting from specific memory safety approaches



```
is_unsafe = call check(ptr, width, witness)
br is_unsafe, label fail, label ok
```

```
ok: store x ptr
    ...
```

```
fail: call report_violation()
      unreachable
```

Figure 1. Scheme of an in-bounds check

allows us to, e.g., define an approach-independent optimization to eliminate redundant checks that filters instrumentation targets. Overall, this common infrastructure enables a fair comparison between the implemented approaches.

3.1 Shared Aspects of Instrumentations

Table 1 shows which code locations are instrumented (Instrumentation Target), for what purpose (Task), and how Low-Fat Pointers and SoftBound instrument them. Our implementation operates on the *Intermediate Representation* (IR) of the LLVM compiler framework [19]. We describe the propagation and checking with (simplified) IR syntax.

The IR represents memory accesses with load and store instructions. At these locations, we want to ensure that the addressed memory is valid to access. However, pointers in LLVM IR, as in C, do not provide their bound values, which we need to ensure safety. Hence, we propagate these values from the allocation to the check location. We call the values that propagate a pointer's bounds a *witness*. The propagation is achieved by code instrumentation, i.e., inserting code at compile time that provides bound values at runtime.

Checks follow the form shown in Figure 1. A `check` function is called, and depending on its outcome, the access is made or an error is reported. As memory accesses can affect multiple bytes, the check needs to ensure that the entire width of the access is part of the allocation. In the following, we describe how to determine the witness for the check.

Allocations are the most overt pointer source, be it as global variables (e.g., `@global_ptr`), on the stack (via LLVM's `alloca` instruction), or on the heap (e.g. via `malloc`). We capture their size information directly as a witness.

Unfortunately, it is not generally possible to determine the allocation for a given pointer at compile time. Hence, we have to instrument additional code locations to determine the corresponding allocation. The first such instruction in Table 1 is a `phi`. In SSA-form programs, ϕ-nodes choose one of the arguments based on which control-flow edge is taken to the node. In our example, when control flows to the `phi` node from block `%b1`, `ptr` is set to `ptr1`, and to `ptr2` when the predecessor is block `%b2`. For every `phi` operating on pointers, additional (witness) `phi` nodes are placed. They propagate metadata for the incoming pointers, e.g., with a `phi` for the pointer base. Analogously, we place additional selects for `select` instructions that operate on pointers. In IR, pointer arithmetic translates to `gep` (aka *get element pointer*)

```
int checkSB(ptr_t ptr, size_t width,
            ptr_t base, ptr_t bound) {
  return ptr < base || ptr > bound - width;
}
```

Figure 2. SoftBound in-bounds check

instructions. They provide a way of indexing into `src_ptr` with indices i_x. Hence, `ptr` inherits the bounds of `src_ptr`.

Compared to these cases, witness propagation becomes more tricky when pointers are loaded from memory, returned from a function, or are function arguments. Here, memory safety instrumentations rely on an approach-specific invariant that describes how to derive the witness. This invariant is set up at stores of pointers to memory, returns of pointer values, and function calls with pointer arguments.

We discuss more concretely how the two approaches ensure that a program is memory safe next.

3.2 SoftBound

SoftBound uses base and bound pointers of an allocation in its in-bounds check, as Figure 2 shows. The check compares the pointer to the lower bound, and the pointer plus width to the upper bound.

For SoftBound, a witness is therefore a pair of base and bound values. When an allocation is encountered, the base pointer is the value returned by `malloc` or `alloca`. The bound is determined by adding the size, which these operations receive as argument, to the base. Globals have a statically known size, which provides the same information.

Whenever possible, SoftBound propagates base and bound values alongside the pointer through the program. For `select` and `phi` instructions operating on pointers (cf. Table 1), it therefore inserts two instructions: one for the base value and one for the bound value. `gep` instructions require no additional code: Their witness is the same the `src_ptr`'s.

Witness propagation is more involved when a pointer escapes the function. For every store of a pointer value to memory, SoftBound tracks the pointer's witness in a *trie data structure* [24, 25, 27]. The address where the pointer is stored is the index for the trie [24, Figure 3]. When a pointer value is loaded from memory, the bounds are taken from the trie.

A *shadow stack* propagates bound information across function calls [25]. It is allocated at program start and operated in sync to the program's call stack. When calling a function with pointer arguments, their base and bound values are pushed to the stack. The callee then relies on the invariant that the shadow stack holds these bounds. Pointers returned from functions are also communicated via this stack.

3.3 Low-Fat Pointers

The core idea of Low-Fat Pointers is to derive base and size of the allocation from the pointer value. They employ a custom memory allocation scheme that groups allocations of similar

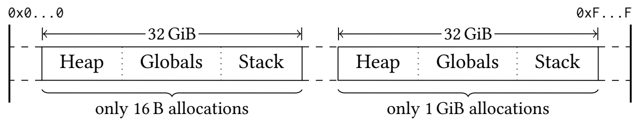


Figure 3. Address space organization for Low-Fat Pointers

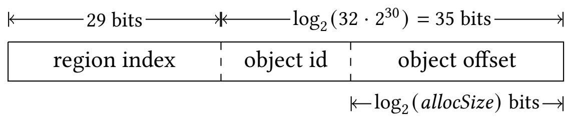


Figure 4. Bit fields of a low-fat pointer value

```
int checkLF(ptr_t ptr, size_t width, ptr_t base) {
  size_t region_idx = lf_region_index(base);
  size_t alloc_size = lf_alloc_size(region_idx);
  return (ptr - base) > (alloc_size - width);
}
```

Figure 5. Low-Fat Pointers in-bounds check

size into regions of the virtual address space. Figure 3 shows these regions. Each region is dedicated to a predetermined allocation size to which allocations are padded if necessary.

With this partitioned virtual address space, a pointer's value suffices to determine the size and the base pointer of the object it is pointing to.[2] Conceptually, a low-fat pointer value consists of three bit fields, shown in Figure 4. The region index in the most significant bits determines in which region the object lies, and therefore the object size S. The remaining bits identify the pointed-to object in the region and the offset of the pointer into this object. The sizes of these last two bit fields depend on the region: In a region with objects of size S, the offset part has enough bits to represent S different offsets, i.e., enough bits to address any byte in the object. Masking away the least significant bits that constitute the object offset yields the object's base pointer.

Reconsider Table 1. One might wonder why we need to propagate witnesses when allocation base and size can be derived from the pointer value. The issue is that this yields the size and base of the object the pointer currently points to. When a pointer is dereferenced after arithmetic, we need to check that it still points to the same object as before. Hence, a Low-Fat Pointers witness for a pointer is the base pointer of the allocation that it points to. Figure 5 shows how to validate a memory access with this Low-Fat Pointers witness.

Low-Fat Pointers rely on the invariant that pointers originating from a different function or from memory are in bounds.[3] Hence, these pointers can be used to derive the correct base value. When pointers escape from a function through stores, calls, or returns, Low-Fat Pointers establish this invariant with an additional in-bounds check.

The initial work on Low-Fat Pointers [10] only supported heap allocations, and was later extended with protection for stack variables [12] and global variables [11].

4 Applicability and Safety Evaluation

Fundamentally, both approaches give wide-ranging spatial safety guarantees: They often prevent out-of-bounds pointers from manipulating other allocations. A distinguishing limitation is that Low-Fat Pointers rely on padded allocations. Consequently, memory accesses that go past the original allocation size into the padding will not be reported. While no other object can be accessed using the pointer, this can leave problems of programs undetected. In contrast, SoftBound uses the original allocation bounds and hence also reports these errors. We discuss the implications of the different design decisions in both approaches and their impact on security and applicability in the following.

4.1 LLVM IR vs. C Code

Both approaches work on LLVM IR rather than C. The LLVM framework allows to easily write compiler passes and its IR provides, compared to C, a fewer language constructs that need to be supported. However, this comes at the cost of potentially reduced bug-finding capabilities: As described by Rigger et al. [29] and Chen et al. [5], some bugs vanish when C is translated to IR, even without optimizations (at optimization level `-O0`). By design, memory safety tools can only provide safety for the IR program they sanitize, not the C program it once was. In the next sections, we highlight cases where the decision to use LLVM impacts our observations.

4.2 Out-of-Bounds Pointer Arithmetic

The C language standard [15] states in its undefined behavior section (J.2), that the program behavior is undefined if:

> "Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6)."

While the language semantics are clear, programmers' expectations often differ. Memarian et al. [21] find in a survey with 323 C experts that 73% of the participants believe that ouf-of-bounds pointer arithmetic works, as long as the pointer is brought back in bounds before accessing memory. This aligns with our findings (see Section 5.1.1) and the findings by Chisnall et al. [6], who found out-of-bounds pointers in 7 out of 15 popular Linux C packages that they investigated.

SoftBound checks that pointers are within bounds upon access, and thus aligns with programmer expectations by not reporting the creation of out-of-bounds pointers as an error. Low-Fat Pointers on the other hand have to perform an in-bounds check to establish their in-bounds invariant,

[2]We restrict our explanation and evaluation to power-of-two sizes, as these performed best in the original evaluation [10].
[3]Allocations are padded by one byte to support one-past-the-end pointers. The behavior of any other out-of-bounds pointer in C is undefined. [15]

for example when a pointer is handed over to a function (cf. Table 1). In general, it is not an option to omit this check to trade safety guarantees for applicability: If a program passes an out-of-bounds pointer to another function, and this function does arithmetic to access the object in bounds, Low-Fat Pointers report an error. The reason is that they check if the incoming and the dereferenced pointer point to the same object, which is not the case for out-of-bounds pointers that are brought back in bounds again. Therefore, the case that programmers expect to work is not supported.

An additional concern is that the semantics of LLVM IR define out-of-bounds pointer arithmetic [37, Section "Pointer Aliasing Rules"]. Transformations in the LLVM pipeline before the instrumentation can thus soundly introduce out-of-bounds pointers where there were none in the original C program, causing spurious errors with Low-Fat Pointers.

Related memory safety approaches such as CRED [30] and Baggy Bounds [1] add costly adaptations to their approaches to support out-of-bounds pointer arithmetic.

4.3 External Libraries

Some programs are linked against libraries that are unavailable for recompilation, e.g., because they are proprietary. Neither discussed approach protects accesses in such libraries.

However, Low-Fat Pointers can still protect most of the recompilable code out-of-the-box in this situation. Heap allocations of the external library automatically use the low-fat `malloc`, and hence fulfill size and alignment requirements. When pointers to these allocations are used in the instrumented code, they are checked for out-of-bounds errors. Global variables and stack allocations of the external library are unprotected. Their addresses are not in the low-fat regions, which Low-Fat Pointers handle by assuming wide bounds, allowing accesses to all memory. E.g., many applications access the output streams of the C Standard Library, `stderr` and `stdout`, which are global variables in the Standard Library. While these accesses are unprotected, they do not result in applicability issues for Low-Fat Pointers.

This is different for SoftBound. Linking against an uninstrumented library requires additional programmer effort if the library communicates pointer values to the instrumented code. If a function returns a pointer, SoftBound assumes that its bounds are on the shadow stack. However, this is not the case for uninstrumented functions. The outdated bounds on the shadow stack usually do not match the allocation accessed by the returned pointer, and hence a violation for a safe access may be reported. Alternatively, an access to the returned pointer can be wrongly classified as safe if the bounds on the shadow stack happen to belong to the accessed object. The solution for interfacing with uninstrumented libraries are *wrappers* for functions that return pointers or manipulate in-memory pointers. The function wrapper updates SoftBound's metadata and calls the original function.

```
void *sb_memcpy(void *dest, void *src, size_t n) {
  void *dest_bs = lookup_bs(1); // get "dest" bounds
  void *dest_bd = lookup_bd(1); // from shadow stack
  check_abort(dest, n, dest_bs, dest_bd);
  check_abort(src, n, lookup_bs(2), lookup_bd(2));
  void *ret_ptr = memcpy(dest, src, n);
  if (n > 0) { copy_metadata(dest, src, n); }
  store_bs_bd_ret(dest_bs, dest_bd);
  return ret_ptr;
}
```

Figure 6. SoftBound wrapper for `memcpy`

Figure 6 shows the wrapper for `memcpy`. The function `copy_metadata` copies the bounds for all pointers in `src` to the metadata space of `dest`. `store_bs_bd_ret` stores the bound values of the returned pointer to the shadow stack. SoftBound additionally uses the wrapper functions to ensure the safety of the called function. The wrapper checks that the given allocations are large enough to load and store the required `n` bytes by calling `check_abort`. This function checks for an out-of-bounds access and aborts if one is detected.

This additional use of the wrapper functions results in better safety guarantees. Low-Fat Pointers can be extended to use wrappers with additional safety checks, but they do not require wrappers to be applicable when linking against uninstrumented libraries.

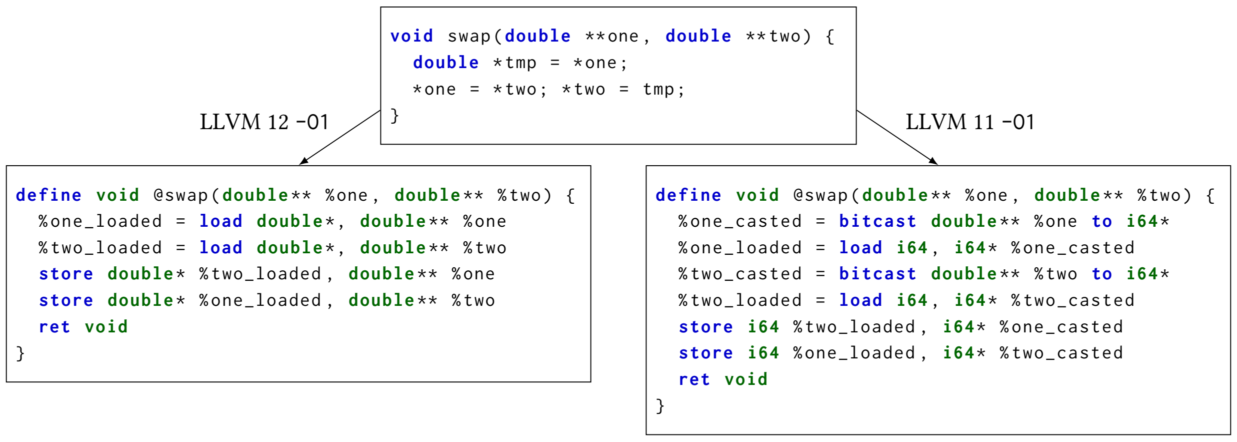
SoftBound has an additional issue that arises when linking with uninstrumented libraries or in the context of separate compilation: C allows programmers to declare global arrays whose definition is in a different translation unit without size information. When a C file with such a declaration is compiled, SoftBound cannot derive the bounds of the array. It can use NULL bounds, likely resulting in spurious violation reports, or wide bounds, losing the ability to detect out-of-bounds accesses. Proper solutions (which require manual effort) are to always declare the arrays with size information, or link all files together before applying SoftBound.

4.4 Integer to Pointer Casts

Integer to pointer casts pose a problem for both approaches.

Low-Fat Pointers introduce an in-bounds check when a pointer is casted to an integer, and rely on the invariant that it still points to the same object when casted back. However, no instrumentation prevents corruption in the meantime. Thus, programs that cast integers to pointers can be unsafe, even if all code is instrumented and all allocations are low-fat.

SoftBound can handle integer to pointer casts in several ways: One is to use NULL bounds for pointers casted from integers. This causes error reports when they are dereferenced. However, C and LLVM allow casting a pointer to an integer, back to a pointer, and then accessing it. Thus, this solution overly restrictive. Due to SoftBound's explicit pointer bounds, another option is to use wide bounds for pointers



```
void swap(double **one, double **two) {
  double *tmp = *one;
  *one = *two; *two = tmp;
}
```

```
define void @swap(double** %one, double** %two) {
  %one_loaded = load double*, double** %one
  %two_loaded = load double*, double** %two
  store double* %two_loaded, double** %one
  store double* %one_loaded, double** %two
  ret void
}
```

```
define void @swap(double** %one, double** %two) {
  %one_casted = bitcast double** %one to i64*
  %one_loaded = load i64, i64* %one_casted
  %two_casted = bitcast double** %two to i64*
  %two_loaded = load i64, i64* %two_casted
  store i64 %two_loaded, i64* %one_casted
  store i64 %one_loaded, i64* %two_casted
  ret void
}
```

Figure 7. Translation of the swap program with clang -O1 for x86_64 with LLVM 12 and LLVM 11

casted from integers. Accessing the resulting pointer can then cause false negatives, as with Low-Fat Pointers.

One last option for both approaches is to provide a witness manually. This improves the safety guarantees by restricting subsequent accesses to the annotated bounds. However, manual annotation is tedious and prone to errors.

One might wonder if these casts are widely used in C programs and advise to rewrite the program to avoid them. While we do not oppose this, it does not resolve the issue for the memory safety instrumentations. LLVM's optimizations introduce such casts even when the C program had none. In addition, LLVM has bitcasts, which can have similar effects as integer-to-pointer casts.

Consider the C swap function in Figure 7 and its two translations to LLVM IR. The swap function loads the pointer value from the first pointer argument, one, and stores it in a temporary variable. Then, it loads the value from its second argument, two, and stores it into one. Lastly, it writes the temporarily stored value into two.

The left IR translation loads the pointer values from both arguments and stores each back into the opposite one. Both instrumentations can set up their invariants at the pointer stores. The translation on the right however obfuscates the stores of pointer values. It first casts the double ** to an i64 * pointer (as the compiler knows that the system has 64-bit-wide pointers). Then, it loads the pointer values as an integers and stores them to the respective memory locations.

With Low-Fat Pointers, no safety check is placed, and, when loaded later on, the pointer is assumed valid anyway. In this example, no harm is caused as the pointer value was not changed. However, this need not be the case in general.

The consequences for SoftBound are even more severe: As only pointer loads and stores are instrumented, this store of an integer will not update the trie data structure. The bounds for the stored values are now outdated, and, when loading the pointer again, the wrong bounds are loaded. In a program without memory errors, loading the double value later on is wrongly reported as a violation. A false negative occurs when the pointer is moved out-of-bounds and into the object to which the old bounds belong.

The translations are not contrived, they are generated by different versions of LLVM (e.g., with 11 and 12). The simple swap C function can thus cause the instrumentations to miss errors or to report spurious ones.

4.5 Byte-Wise Copying of Pointers

A similar metadata update issue as for the swap program in Figure 7 for SoftBound occurs when memory content is copied byte-by-byte. The C Standard [15, Section 6.3.2.3:7] allows casting any pointer to char * to inspect an object's byte representation. This causes issues for SoftBound if the copied object contains pointer values. Since the pointer is not copied as a whole, the copy goes unnoticed. The metadata for the pointer at the new in-memory location is therefore missing, causing the same unexpected behavior as described in Section 4.4. Again, this poses no problem for Low-Fat Pointers.

4.6 Safe Dereferences

There are cases where both memory safety instrumentations have to assume that a pointer can access every memory location. As discussed in Section 4.3, this is the case for SoftBound when files are compiled separately and arrays are declared as external without size information. Low-Fat Pointers do not have bounds information for global or stack variables of external libraries. In addition, if a low-fat region runs out of memory or an object that is larger than the largest supported size is allocated, Low-Fat Pointers resort to the standard allocator and hence generate non-low-fat pointers.

Neither of the two approaches gives a quantitative evaluation of how many accesses remain unchecked because of the intrinsic limitations of the respective approach. We report the results of this evaluation on our benchmarks (their selection is discussed in Section 5.1.1) in Table 2. For each

Table 2. Number of unsafe dereferences in %, **bold benchmarks** contain size-zero array declarations, an asterisk (*) indicates zero unsafe accesses

Benchmark	SB	LF	Benchmark	SB	LF
164gzip	61.71	0.00	401bzip2	0.00*	0.00
177mesa	0.00*	1.57	429mcf	0.00*	54.08
179art	0.00*	0.00	**433milc**	0.00*	0.00
181mcf	0.00*	0.00	**445gobmk**	0.66	0.06
183equake	0.00*	0.00	**456hmmer**	0.00	0.08
186crafty	0.00*	0.00	**458sjeng**	0.00	1.17
188ammp	0.00*	0.24	462libquant	0.00*	0.00
197parser	0.27	7.14	464h264ref	0.00*	2.84
256bzip2	0.00*	0.00	470lbm	0.00*	0.00
300twolf	0.37	2.08	482sphinx3	0.00*	0.00

benchmark, we can see the percentage of access checks that use wide bounds for SoftBound and Low-Fat Pointers. The 0% entries marked with an asterisk (*) had not a single check with wide bounds. In both approaches, the vast majority of accesses in the benchmarks are checked.

However, there are exceptions: We marked benchmarks with size-less declarations of arrays in blue and bold. `164gzip`, where this feature is widely used, suffers from 62% unchecked accesses with SoftBound. All benchmarks using this feature suffer from at least some unchecked accesses. The only exception is `433milc`, where such an array is declared, but not used in the benchmark run. Some benchmarks also contain integer to pointer casts, for which wide bounds are propagated. With Low-Fat Pointers, the benchmark `429mcf` suffers from 54% unchecked accesses. This benchmark has one allocation that is unchecked because it exceeds the largest region size, in our case 1 GiB.

We can see that programming patterns such as large allocations for Low-Fat Pointers and externally defined arrays without size information for SoftBound weaken the safety guarantees. Both limitations can be overcome, e.g., by adapting the configuration for Low-Fat Pointers, or adding size information for SoftBound. However, this requires thorough inspection of the tools' reports and a fundamental understanding of their inner workings.

4.7 Conclusion: Applicability and Safety

In our experiments, SoftBound often reported false violations due to outdated metadata (cf. Section 4.4 and 4.5). Fortunately, `inttoptr` casts can be detected statically and reported to the tool user as a potential reason for false positives or negatives. Byte-wise copies, however, are hard to debug and hard to find automatically.

As reported by the original papers, SoftBound is very effective to detect spatial safety errors, while Low-Fat Pointers tend to prevent their exploitation, but are often not able to report them. With respect to applicability in the presence of uninstrumented code, Low-Fat Pointers are likely easier to use as they do not require implementing wrappers. However, if the program uses out-of-bounds pointer arithmetic, SoftBound is the better candidate.

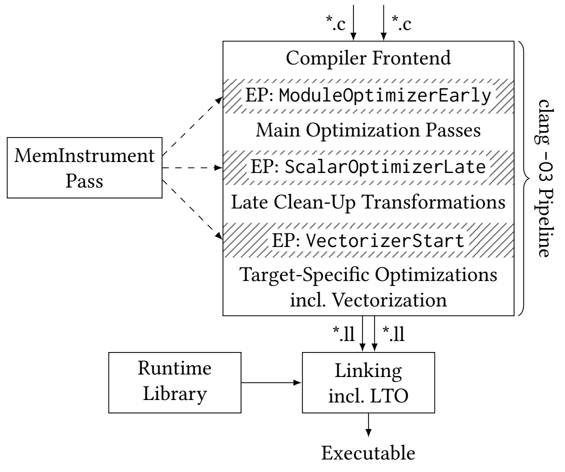


Figure 8. Technical setup (LTO = Link-Time Optimization). The MemInstrument pass can be inserted at any of the highlighted extension points (EP) of clang.

5 Runtime Evaluation

We compare Low-Fat Pointers and SoftBound, implemented in our instrumentation framework MemInstrument.

The implementation for Low-Fat Pointers protects heap allocations [10], as well as stack [12] and global variables [11]. As we are primarily interested in runtime performance, we use powers-of-two allocation sizes, $\{2^4, .., 2^{30}\}$ (16 B to 1 GiB).

During SoftBound's development, the original authors continuously improved the data structures for the bounds metadata [22–25], we chose the most recent ones. Our implementation additionally protects variable argument functions. We do not support vectorized code as extending SoftBound accordingly is not straightforward.

5.1 Setup

Figure 8 shows the evaluation setup. Our framework is an LLVM module pass and can be plugged into the compiler pipeline at various *extensions points*. During compilation, MemInstrument places code for witness propagation and inserts calls to check functions. For SoftBound, functions that require wrapping are replaced by their wrapped counterparts. We compile the files separately, and link them together afterwards with link-time optimizations. Definitions for the inserted calls are in the linked runtime library.

Our approach uses an unmodified version of LLVM 12 [37].

5.1.1 Benchmark Selection. We use the C benchmarks from SPEC CPU2006 [14] and CPU2000 [34]. SoftBound does not support C++ and neither approach supports Fortran; this

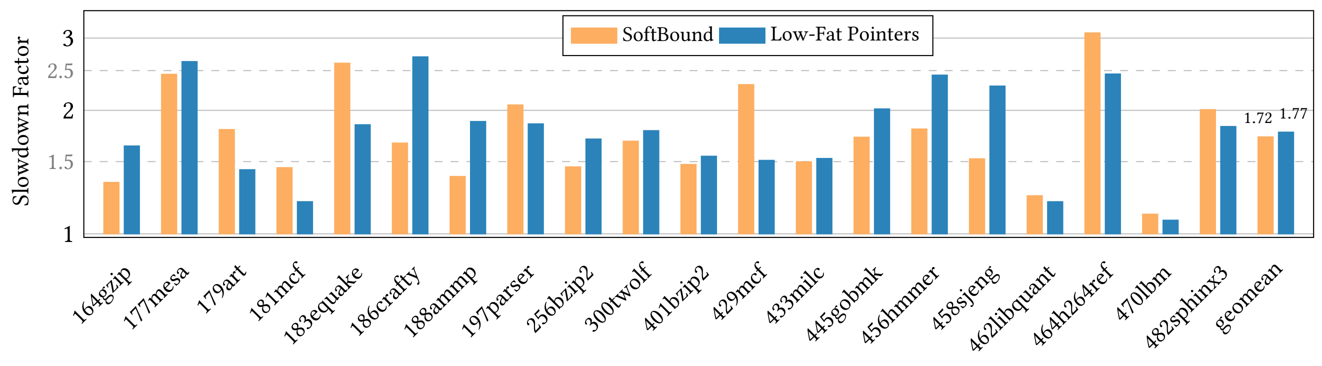


Figure 9. Execution Time Comparison

reduces the number from 45 to 27 benchmarks. We evaluate only the 20 benchmarks that execute successfully with both approaches (cf. Section 5.1.2).

The `perl` (`253`/`400`) and `254gap` benchmarks use pseudo-base-one arrays: They create a pointer one element before the start of an array. This undefined behavior results in violation reports from Low-Fat Pointers. SoftBound reports known violations for `perl`, and none for `254gap`. The `gcc` (`176`/`403`) benchmark uses NULL pointers with large offsets to access memory [18], and performs out-of-bounds pointer arithmetic, both violations of the C Standard, and errors are reported by Low-Fat Pointers and SoftBound. The last two, `175vpr` and `255vortex`, use out-of-bounds pointer arithmetic, which Low-Fat Pointers, but not SoftBound, reports.

5.1.2 Comparability. SoftBound uses function wrappers for the Standard Library to update its metadata. However, these wrappers additionally check the called functions for safety. We disabled these checks, as they could impact the runtime comparability of the approaches (cf. Section 4.3).

The publications presenting SoftBound and Low-Fat Pointers evaluate different benchmarks, and Duck and Yap [10] specifically exclude functions in some benchmarks that otherwise cause errors because of out-of-bounds pointer arithmetic. We have two choices on how to handle such cases: (a) We exclude the same functions for both tools, or (b) we leave these benchmarks out. While (a) is valid to compare the approaches against each other, the performance overhead of an only partially checked benchmark does not give a meaningful overall slowdown value.[4] Whenever failing benchmarks could be fixed with minor source code modifications, we adapted the benchmarks, otherwise we left them out (option (b)).

In `181mcf`, a pointer is stored in a struct member with integer type (cf. Section 4.4). We changed the type to the proper pointer type, and dropped the now unnecessary casts of pointers from/to integers. The problem of byte-wise pointer copies described in Section 4.5 occurs in `300twolf`. We replaced the byte-wise copy by a `memcpy`. Both cases lead to spurious error reports from SoftBound because it could not properly update its metadata, while Low-Fat Pointers did execute the benchmark without modification. Lastly, we fixed a known out-of-bounds access in `197parser`, and two in `464h264ref`. The out-of-bounds access in `197parser` goes to the allocation padding of Low-Fat Pointers, and hence no error is reported. Low-Fat Pointers report an error on `464h264ref`, SoftBound reports all of these errors.

[4]Note that, compared to unsafe accesses that are checked as discussed in Section 4.6, for excluded functions no code will be inserted at compile time and hence there is a reduced execution time overhead.

5.2 Runtime Comparison

We use identical machines with an Intel Core i9-10900K @ 3.70GHz processor and 64 GB of RAM, running a Ubuntu 20.04.4 LTS (kernel version 5.4.0). We disabled SMT and Turboboost, and use the `powersave` scaling governor to minimize inter-run variation. All reported numbers are the median of 7 samples, executed sequentially. We noticed little inter-run variation.

Figure 9 shows the execution time overheads of SoftBound and Low-Fat Pointers, normalized to the runtime of clang `-O3` (1×). Both are optimized with a simple dominance-based check elimination described in Section 5.3, and inserted into the compiler pipeline at the extension point `VectorizerStart`. We can see that their mean slowdown is comparable, 1.74× for SoftBound and 1.77× for Low-Fat Pointers. However, individual benchmarks vary widely, and sometimes one or the other approach performs better.

The reasons for this lie in their design: SoftBound performs worse than Low-Fat Pointers on `equake`. This benchmark has a particularly hot loop that loads pointer values from memory. SoftBound has to look up the bounds for the pointers in the trie data structure, while Low-Fat Pointers only recalculate the base pointer.

In other benchmarks, such as `crafty`, SoftBound outperforms Low-Fat Pointers. We can see from the checks in Figure 2 and Figure 5 that a SoftBound check requires fewer instructions. Hence, it is cheaper to execute.

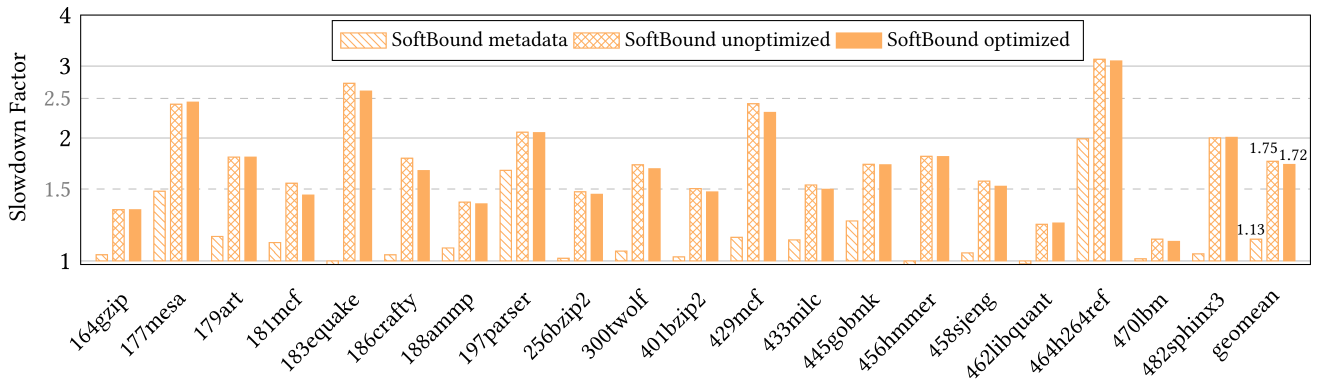


Figure 10. Execution Time Comparison of SoftBound optimized, unoptimized, and only metadata propagation

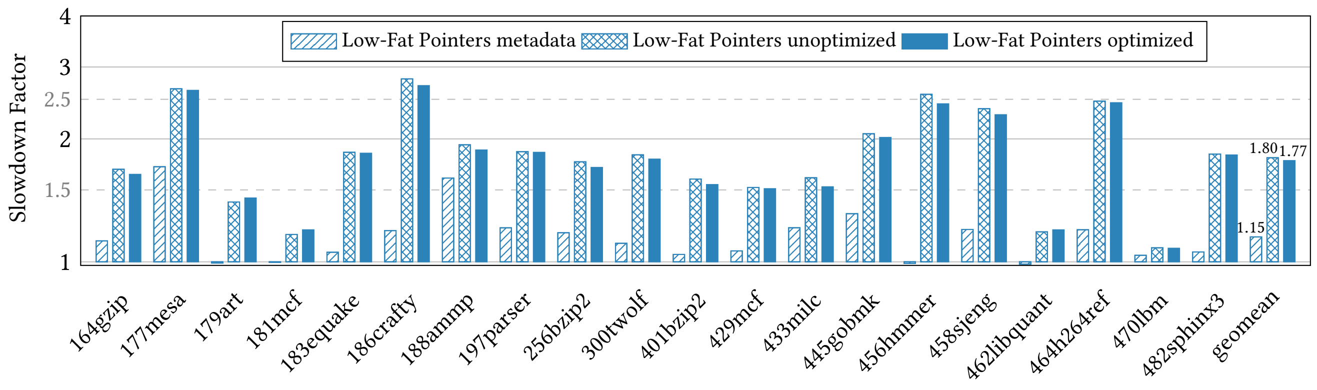


Figure 11. Execution Time Comparison of Low-Fat Pointers optimized, unoptimized, and only metadata propagation

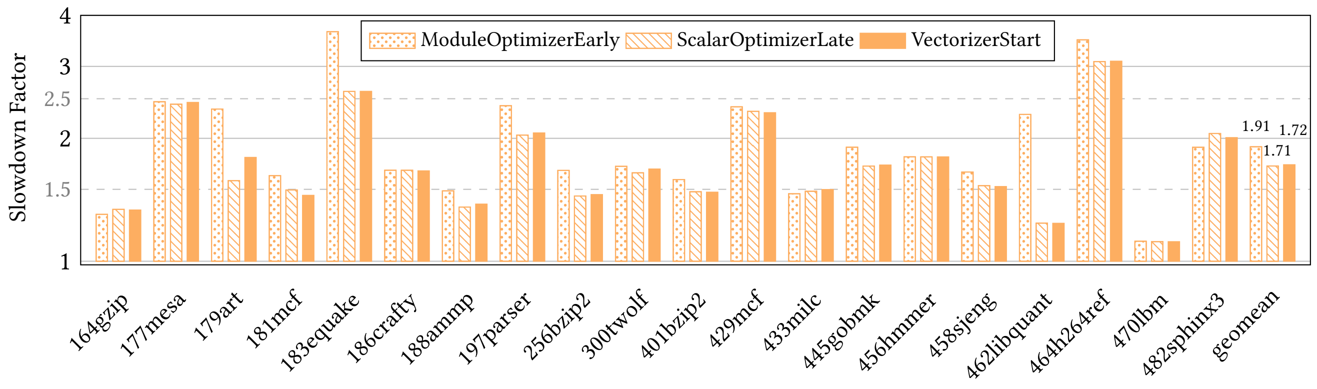


Figure 12. Impact of Compiler Pipeline Extension Points on SoftBound

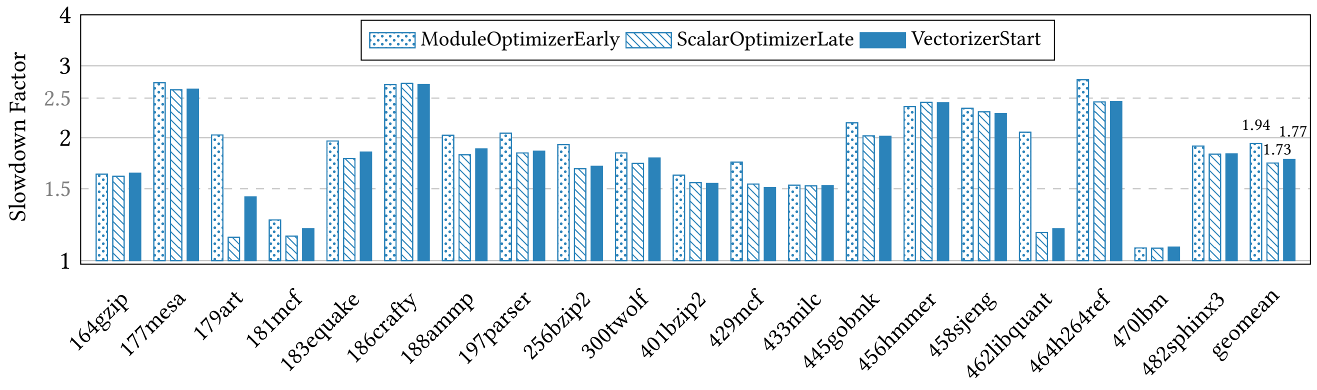


Figure 13. Impact of Compiler Pipeline Extension Points on Low-Fat Pointers

Our results do not fully reproduce the results of the original papers, and this has various reasons: We have a different set of benchmarks, different hardware, and different compiler versions than both approaches. Low-Fat Pointers used `-O2` and no link-time optimizations. SoftBound linked before instrumenting the code. Evidently, these differences matter, which supports the need for a common framework to compare different approaches.

5.3 Impact of Optimizations

We implemented a simple optimization to eliminates checks that is frequently described in literature [1, 10, 23]. Whenever two accesses to the same memory location are made, and one dominates the other, the dominated check is redundant. While this removes a significant number of checks, between 8% for `177mesa` and 50% for `256bzip2`, the runtime impact is minor. Our results are in Figure 10 for SoftBound and in Figure 11 for Low-Fat Pointers. As remarked by Duck and Yap [10], the compiler can optimize away these checks on its own, and it seems to do so effectively in our setting.

5.4 Invariant Costs

We measured the impact of establishing the invariant of the instrumentations, without executing access checks. Figure 10 shows low overhead for SoftBound in many cases, but invariants can also dominate the overhead in cases like `197parser` and `464h264ref`. This overhead is largely caused by maintaining bounds for in-memory pointers in the trie data structure. However, we see that `equake` has low overheads while we claimed earlier that it has a metadata load within its hot loop. The reason for this phenomenon is that when bounds metadata is loaded but unused afterwards, as in this configuration without checks to use it, the compiler optimizes the load away. Therefore, the numbers here underapproximate the cost to propagate the metadata of SoftBound.

The Low-Fat Pointers invariant ensures that pointers stored to memory, returned from a function or passed to another function are in bounds. The overhead of these checks is shown in Figure 11.

5.5 Impact of the Compiler Pipeline

Lastly, we evaluated the runtime impact of instrumenting at different points in the compiler pipeline. Figure 12 and Figure 13 show the results. While both are comparable on `ScalarOptimizerLate` and `VectorizerStart`, they are slow at the early extension point `ModuleOptimizerEarly`, where the code is instrumented before the main optimizations (cf. Figure 8). Memory safety checks are very effective at preventing optimizations. The checks may abort the program, and the compiler usually cannot prove that the abort is not executed, hence optimizations like code motion and loop transformations are blocked. At later extension points, the number of memory accesses is reduced, and so is the number of checks, resulting in better performance overall.

The gap between the early and later extension points show how problematic it is to compare memory safety approaches on different grounds: If you pick the numbers for one approach on the early extension point and choose the late one for the other, you can conclude that either approach is around 30% faster than the other one.

6 Related Work

Szekeres et al. [36] give a comprehensive overview of different security vulnerabilities. They discuss what components are required to execute an attack on a program, with memory safety violations being the root to all of them.

Song et al. [33] discuss sanitizers for many kinds of undefined behavior, including spatial memory safety violations, signed integer overflow errors, and bad type casts. They compare safety guarantees, space and time overheads, and discuss the underlying mechanisms. While they feature an evaluation aiming to reproduce the results for available tools, they do so by using the publicly available versions. They were able to execute Low-Fat Pointers on the SPEC CPU2006 benchmarks, and report the tool as functional with roughly the execution time overhead stated by the authors. However, they were unable to successfully execute SoftBound and omitted it for their evaluation.

Llorente-Vazquez et al. [20] provide an index for the many possible attacks enabled by memory safety errors, and defenses at different levels. This includes strategies to detect memory safety errors like static analysis, fuzzing, symbolic execution, and sanitizers.

All three surveys [20, 33, 36] compare a broad range of approaches on a high level. In contrast, the goal of our work is to compare approaches in detail. We evaluate the security guarantees and applicability in depth, and give a fair comparison on the grounds of a common framework.

7 Conclusion

We presented an in-depth comparison of the two prominent spatial memory safety checkers, Low-Fat Pointers and SoftBound, implemented on common grounds to allow a fair comparison. Our evaluation shows comparable runtime overheads on average, but one or the other gives a better performance depending on the characteristics of the program.

SoftBound and Low-Fat Pointers are very effective to prevent the exploitation of spatial memory safety errors. However, both tools have several limitations in their usability. For example, the bounds metadata of SoftBound can get outdated, leading to spurious error reports. Low-Fat Pointers report out-of-bounds pointer arithmetic as an error in some situations, counter to the common expectation of programmers that only out-of-bounds accesses are reported as error. Both issues are amplified by compiler optimizations and the semantical differences of C and LLVM IR.

Our findings demonstrate problems in the applicability of both approaches that future research has to solve in order to enable widespread use of memory safety measures in practice. In addition, there is a significant performance gap between mere bounds metadata propagation and full checking. We see the potential for further check optimizations here, to ultimately enable the usage in software releases, rather than only during development and debugging.

We open-source our LLVM-based MemInstrument framework to enable users to compare the tools on their programs, and to provide researchers who explore new memory safety approaches with an extensible base and fair comparison options to existing approaches.

A Artifact Appendix

A.1 Abstract

The artifact consists of a Vagrant VM with pre-installed dependencies for MemInstrument, as well as our MemInstrument development. Since the SPEC benchmarks evaluated in the paper are proprietary, we provide execution instructions for our MemInstrument compiler plugin which can be used to instrument arbitrary C programs. We provide around 200 small C programs which can be executed to verify the functionality of the artifact.

The source code available on GitHub can be used to extend MemInstrument. The archived artifact contains a copy of this repository.

A.2 Artifact check-list (meta-information)

- **Algorithm: MemInstrument**
- **Program: Small C programs**
- **Run-time environment: Linux, Vagrant, VirtualBox**
- **Hardware: x86-64**
- **Execution: less than 1 minute**
- **How much disk space required (approximately): 15 GiB**
- **Publicly available: https://github.com/cdl-saarland/MemInstrument**
- **Code licenses: NCSA (University of Illinois/NCSA Open Source License)**
- **Archived: https://doi.org/10.5281/zenodo.13345361**

A.3 Description

A.3.1 How delivered. Use the link provided in the above check-list under *Archived* to find a pre-built MemInstrument in a Vagrant VM.

A.3.2 Hardware dependencies. The artifact execution requires a x86-64 processor.

A.3.3 Software dependencies. We have tested the artifact on Linux only, it requires Vagrant and VirtualBox to run the Vagrant VM.

A.4 Installation

The artifact contains an `artifact_usage.md` file, which describes the installation steps.

The source code repository contains a README.md file for setup and testing.

A.5 Evaluation and expected result

The testing, also described in the `artifact_usage.md` file, executes numerous tests with MemInstrument. These tests check that programs which contain memory safety violations such as heap, stack or global variable out-of-bounds accesses are correctly identified and that no error is reported on C programs without out-of-bounds accesses. I.e., a program is instrumented with Low-Fat Pointers or SoftBound and the resulting program is executed. The tests automatically validate against the expected outcome.

A.6 Notes

The basic command line flags for compiler and linker of the approaches are described in Section A.4, this section describes the additionally required flags to acquire the results in Figure 9 to Figure 13.

SoftBound configuration basis:

```
-mi-config=softbound
-mi-sb-size-zero-wide-upper
-mi-sb-inttoptr-wide-bounds
```

Low-Fat Pointers configuration basis:

```
-mi-config=lowfat
-mi-lf-transform-common-to-weak-linkage
```

Common (additional) configuration:

- Always (to ignore inline assembler):
  `-mi-policy-ignore-inline-asm`
- For the *metadata* configuration:
  `-mi-mode=geninvariants`
- For the *optimized* configuration:
  `-mi-opt-dominance`

In order to change the location of MemInstrument in the compiler pipeline (as done for Figure 12 and Figure 13), change `meminstrument/lib/RegisterPasses.cpp:104` to

- `EP_ModuleOptimizerEarly`,
- `EP_ScalarOptimizerLate`, or
- `EP_VectorizerStart`.

`EP_ModuleOptimizerEarly` is the default in the artifact. Recompile MemInstrument for the change to take effect.

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *SSYM '09* (Montreal, Canada). USENIX Association, USA, 51–66. https://dl.acm.org/citation.cfm?id=1855772

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (Eds.). ACM, 290–301. https://doi.org/10.1145/178243.178446

[3] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 213–223. https://doi.org/10.1109/CGO.2011.5764689

[4] Nathan Burow, Derrick Paul McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim (Eds.). ACM, 381–392. https://doi.org/10.1145/3196494.3196540

[5] Zhe Chen, Chong Wang, Junqi Yan, Yulei Sui, and Jingling Xue. 2021. Runtime detection of memory errors with smart status. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 296–308. https://doi.org/10.1145/3460319.3464807

[6] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2694344.2694367

[7] MITRE Corporation. 2023. Most Stubborn Weaknesses in the CWE Top 25, Common Weakness Enumeration CWE. https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html. Accessed: 2024-05-30.

[8] MITRE Corporation. 2023. Top 25 Most Dangerous Software Errors 2023, Common Weakness Enumeration CWE. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. Accessed: 2024-05-30.

[9] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*. 162–171. https://doi.org/10.1145/1134285.1134309

[10] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction - CC 2016*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[11] Gregory J. Duck and Roland H. C. Yap. 2018. An Extended Low Fat Allocator API and Applications. *CoRR* abs/1804.04812 (2018). arXiv:1804.04812 http://arxiv.org/abs/1804.04812

[12] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/stack-object-protection-low-fat-pointers/

[13] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*. IEEE Computer Society, 53–60. https://doi.org/10.1109/SecDev.2018.00015

[14] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[15] ISO/IEC. 2018. ISO/IEC 9899:2018 Information technology — Programming languages — C. https://www.iso.org/standard/74528.html. Accessed: 2024-05-30.

[16] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, Carla Schlatter Ellis (Ed.). USENIX, 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html

[17] Richard W. M. Jones and Paul H. J. Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the Third International Workshop on Automated Debugging*. 13–26.

[18] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks without the Checks. In *EuroSys '18* (Porto, Portugal). ACM, 14 pages. https://doi.org/10.1145/3190508.3190553

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04* (Palo Alto, California). IEEE Computer Society, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[20] Oscar Llorente-Vazquez, Igor Santos, Iker Pastor-Lopez, and Pablo Garcia Bringas. 2022. The Neverending Story: Memory Corruption 30 Years Later. In *14th International Conference on Computational Intelligence in Security for Information Systems and 12th International Conference on European Transnational Educational (CISIS 2021 and ICEUTE 2021)*, Juan José Gude Prego, José Gaviria de la Puerta, Pablo García Bringas, Héctor Quintián, and Emilio Corchado (Eds.). Springer International Publishing, Cham, 136–145. https://doi.org/10.1007/978-3-030-87872-6_14

[21] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 1–15. https://doi.org/10.1145/2908080.2908081

[22] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *SNAPL 2015*, Vol. 32. Dagstuhl, Germany, 190–208. https://doi.org/10.4230/LIPIcs.SNAPL.2015.190

[23] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI '09* (Dublin, Ireland). ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, Jan Vitek and Doug Lea (Eds.). ACM, 31–40. https://doi.org/10.1145/1806651.1806657

[25] Santosh Ganapati Nagarakatte. 2012. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. Ph. D. Dissertation. University of Pennsylvania, USA. Advisor(s) Martin, Milo M.

[26] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the Principles of Programming Languages 2002*. Portland, OR USA, 128–139. https://doi.org/10.1145/565816.503286

[27] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007*, Chandra Krintz, Steven Hand, and David Tarditi (Eds.). ACM, 65–74. https://doi.org/10.1145/1254810.1254820

[28] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2 (2018), 28:1–28:30. https://doi.org/10.1145/3224423

[29] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 377–391. https://doi.org/10.1145/3173162.3173174

[30] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2004/practical-dynamic-buffer-overflow-detector/

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 309–318.

[32] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 17–30. http://www.usenix.org/events/usenix05/tech/general/seward.html

[33] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)* (San Francisco, CA, USA). IEEE, 1275–1295. https://doi.org/10.1109/SP.2019.00010

[34] Standard Performance Evaluation Corporation SPEC. 2000. SPEC CPU 2000 benchmarks. https://www.spec.org/cpu2000/.

[35] Standard Performance Evaluation Corporation SPEC. 2017. SPEC CPU 2017 benchmarks. https://www.spec.org/cpu2017/. Accessed: 2024-05-30.

[36] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Xiaodong Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62. https://doi.org/10.1109/SP.2013.13

[37] The LLVM Team. 2021. LLVM Language Reference Manual (Version 12). https://releases.llvm.org/12.0.0/docs/LangRef.html. Accessed: 2024-05-30.

[38] Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuandong Li, and Zhiqiang Zuo. 2023. Catamaran: Low-Overhead Memory Safety Enforcement via Parallel Acceleration. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 816–828. https://doi.org/10.1145/3597926.3598098

B Appendix

In various papers in the area of spatial memory safety, overflows within `structs` are discussed. We want to give some details how Low-Fat Pointers and SoftBound can deal with them and share some thoughts on the topic[5].

```
struct simple_pair { int x; int y; } P;
int main() { print(&P.y - 1); }
```

```
%simple_pair = type { i32, i32 }
@P = global %simple_pair zeroinitializer
define i32 @main() {
  call void @print(i32* getelementptr
  (%simple_pair, %simple_pair* @P, i64 0, i32 0))
  ret i32 0
}
```

Figure 14. Intra-object overflows

B.1 Intra-Object Overflows

Consider the program in Figure 14, defining the structure `simple_pair` with members `x` and `y`. The main function calls `print` with the pointer argument `&P.y-1`. The C Standard [15, 6.7.2.1:14] does not guarantee what value will be at this address, as the value depends on the implementation-defined padding between struct members. Therefore, memory safety instrumentations aim to report errors for such accesses.

Low-Fat Pointers cannot protect against these so-called *intra-object overflows* by design. SoftBound can narrow its bounds to the struct member and is hence capable of identifying such errors. However, in practice, automatic narrowing of bounds and thereby detecting intra-object overflows is not that simple. Consider Figure 14 again, this time the LLVM IR code on the bottom, which is generated from the C code (with clang `-O1`). The address computation is translated to a `gep` instruction with all-zero indices.[6] This means it refers to `P.x` and the arithmetic on the address of `P.y` has disappeared. Effectively, there is no more issue to be reported here.

Note however that bounds narrowing is hard to automate even at C level. It is guaranteed by the language standard that a struct has the same address as its first member. For our small `simple_pair`, this means that `&P == &P.x`, and a programmer might make use of this property. In this case, narrowing to the bounds of `&P.x` could lead to false positives, while not doing it can lead to false negatives. There is an even more problematic case: When a function receives a pointer to a struct, do you narrow the bounds to the size of the struct? What if this pointer is pointing into an array of structs, and the programmer wants to iterate over the array?

As automated narrowing can only guess the programmer's intention, the detection of intra-object overflows likely requires programmers to give manual hints on which pointer bounds to narrow.

[5] For an overview of other approaches and how they handle them see [33].
[6] It computes the address of the first member (second "0" index) of the first `simple_pair` in the allocation (first "0" index).