In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Granularity Protection

Shengjie Xu
shengjie.xu@mail.utoronto.ca
University of Toronto
Canada

Wei Huang
wh.huang@mail.utoronto.ca
University of Toronto
Canada

David Lie
lie@eecg.toronto.edu
University of Toronto
Canada

ABSTRACT

Programming languages like C and C++ are not memory-safe because they provide programmers with low-level pointer manipulation primitives. The incorrect use of these primitives can result in bugs and security vulnerabilities: for example, spatial memory safety errors can be caused by dereferencing pointers outside the legitimate address range belonging to the corresponding object. While a range of schemes to provide protection against these vulnerabilities have been proposed, they all suffer from the lack of one or more of low performance overhead, compatibility with legacy code, or comprehensive protection for all objects and subobjects.

We present In-Fat Pointer, the first hardware-assisted defense that can achieve spatial memory safety at subobject granularity while maintaining compatibility with legacy code and low overhead. In-Fat Pointer improves the protection granularity of tagged-pointer schemes using object metadata, which is efficient and binary-compatible for object-bound spatial safety. Unlike previous work that devotes all pointer tag bits to object metadata lookup, In-Fat Pointer uses three complementary object metadata schemes to reduce the number pointer tag bits needed for metadata lookup, allowing it to use the left-over bits, along with in-memory type metadata, to refine the object bounds to subobject granularity. We show that this approach provides practical protection of fine-grained spatial memory safety.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**; **Software and application security**; **Systems security**.

KEYWORDS

Memory Safety, Spatial Memory Errors, Buffer Overflows, Subobject Bound Checking, Bound Checking, Tagged-pointer, Object Metadata

ACM Reference Format:
Shengjie Xu, Wei Huang, and David Lie. 2021. 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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3445814.3446761

1 INTRODUCTION

Memory corruption vulnerabilities in programs written in non-memory-safe programming languages like C and C++ have been a research focus for more than a decade [40]. When a pointer is dereferenced, errors, causing memory-safety violations, can result in the unintended disclosure or corruption of memory. Such errors are a powerful primitive for attackers to mount further attacks, such as code reuse attacks [34] and data-only attacks [18]. Memory-safety violations usually result from spatial memory errors, where incorrect pointer arithmetic causes an address to point outside the intended memory range when it is dereferenced.

To detect all spatial memory errors, the value of the pointer must be checked against pointer bounds, which are metadata that describes the memory range over which the pointer may safely point [40]. Traditional fat-pointer schemes are able to maintain full, fine-grained bounds, by either storing these bounds in a separate memory region (i.e., out-of-band), which incurs 50% or higher performance and memory overhead on average with high (sometimes an order of magnitude) variance depending on the program [10, 29, 32]; or *inline* with the pointer, which changes the memory layout and loses pointer-size compatibility with legacy code [42]. Therefore, more recent work explores coarse-grained spatial protection approaches [5, 24, 30, 36, 38], but must still trade-off protection granularity, compatibility, and efficiency. As a result, none of these approaches are able to achieve spatial protection at the same granularity as fat-pointers without losing compatibility or efficiency.

In this work, we propose In-Fat Pointer, a hardware-assisted spatial memory defense that provides fine-grained protection on-par with traditional fat-pointer schemes but preserves compatibility and efficiency. To provide pointer-size compatibility with legacy code without storing pointer metadata in a separate region, In-Fat Pointer employs a *tagged-pointer* scheme, which uses the unused upper bits (which we call the *tag*) on the pointer [24]. Because the size of the tag (usually 16 bits or less) is too small to store full metadata, In-Fat Pointer follows the approach used in previous work [5, 14, 22, 30] which uses the tag to facilitate lookups to per-object metadata stored in memory. However, previous work using this approach alone can only provide *object*-granularity protection and cannot detect *intra-object* overflow as fat-pointer schemes would. Considering the short code example in Listing 1. Such a scheme may detect overflow from `vulnerable` array at Line 2 only if the overflow corrupts memory outside of `struct S`, while a finer-grained scheme providing *subobject*-granularity[1] protection should also detect intra-object overflow that corrupts the `sensitive` subobject at Line 3. To achieve this protection granularity, the pointer

[1]A subobject represents a C struct member, array, or any data field.

```c
struct S {
  char vulnerable[12]; /* attacker can overflow */
  char sensitive[12];
};
```

Listing 1: Simple Intra-object Overflow Example

bounds must be maintained as *per-pointer* metadata[2] so that, for example, when a pointer to `struct S` derives another pointer to `vulnerable`, the bounds of the derived pointer can be *narrowed* to exactly cover `vulnerable` without affecting the source pointer. Therefore, In-Fat Pointer needs to find an efficient way to (1) compactly represent the subobject bounds on pointer tags, and (2) save pointer tag bits from object metadata lookup to be used for bounds narrowing operations.

To address the two challenges, In-Fat Pointer utilizes two key techniques. First, In-Fat Pointer efficiently narrows the object bounds to that of a subobject by (1) identifying which subobject the narrowed pointer points to using per-pointer metadata in the tag and (2) using per-type metadata indexed by subobject to compute the subobject bounds. Second, by designing multiple object metadata schemes with each scheme for one category of objects, In-Fat Pointer can devote fewer bits to object metadata lookup (and therefore more bits for subobject information) without losing efficiency, as opposed to previous approaches that devote all pointer tag bits to object metadata lookup.

To the best of our knowledge, In-Fat Pointer is the first hardware-based spatial memory defense using a tagged-pointer approach that is able to protect against intra-object overflow while maintaining the same pointer size as legacy code. We show that with these approaches combined, In-Fat Pointer can provide spatial memory safety comparable with fat pointer schemes while preserving compatibility and low overhead.

This paper makes the following contributions:

- We present the design and implementation of In-Fat Pointer, a hardware-assisted tagged-pointer scheme that provides subobject-granularity spatial memory protection. We describe how In-Fat Pointer uses hardware-software co-design to enforce fine-grained spatial memory safety with binary compatibility with legacy code and low overhead.
- We describe three complementary metadata schemes used by In-Fat Pointer. We show that by *designing* separate object metadata schemes for each category of objects, the three In-Fat Pointer schemes reduce consumption of precious pointer tag bits without losing efficiency.
- We describe the subobject bounds narrowing mechanism that In-Fat Pointer uses to refine the object bounds to subobject granularity during pointer bounds retrieval.
- We implement an In-Fat Pointer prototype on an FPGA. We evaluate its ability to detect memory safety violations using the Juliet test suite, and performance and memory overhead against a set of application benchmarks.

[2]We use "per-pointer metadata" to refer to metadata on each pointer that can change without affecting the metadata on other pointers. For example, if all pointers to the same object must be associated with identical metadata, the metadata is per-object.

2 BACKGROUND AND RELATED WORK

Because pointer safety cannot always be determined at compile time, all defenses enforcing spatial memory safety introduce additional metadata and dynamic checks into the program to verify that memory accesses are safe. Table 1 summarizes previous work in this area. There is a large design space on possible metadata and assertions, which results in varying protection granularity and limitations, including loss of compatibility and dependence on hardware or software features. We group the related work into (1) fat-pointer schemes, (2) object-based schemes, (3) memory-based schemes, and (4) tagged-pointer schemes in the table according to the subject of introduced in-memory metadata and the use of tagged-pointer. Note that while the use of tagged-pointer is orthogonal to the use of in-memory metadata, all schemes using tagged-pointers in their designs are considered tagged-pointer schemes in this section.

Because the only way to detect all spatial memory errors is to associate pointer bounds to pointers and perform bounds checking [40], and retrieving and maintaining pointer bounds tends to contribute most to the cost of spatial safety enforcement [32], when discussing related works that enforce object- or subobject-granularity protection, we focus on the schemes used for pointer bounds retrieval and maintenance.

2.1 Fat-Pointer Schemes

Traditional fat-pointer schemes maintain *persistent*[3] per-pointer bounds throughout the lifetime of a pointer, and then check the pointer against the bounds when it is dereferenced. Such pointer-based schemes can detect overflows at the subobject level: when a pointer to an object derives a pointer to one of the subobjects, for example, by taking the address of a struct member, the bounds for the derived pointer can be narrowed to stop overflow that would corrupt other struct members. To maintain the bounds (and additional metadata if any) with the pointer, the most straightforward way is to store them along with the pointer itself [9, 42], thus increasing the apparent size of the pointer. For example, CHERI [42] extends each 64-bit pointer to a capability pointer, which includes permissions and bounds information and is 256 bits in the worst case. However, storing metadata with the pointer has two drawbacks: (1) it increases memory overhead as each pointer is increased in size, and (2) it creates binary incompatibility between the instrumented code and legacy code that makes assumptions about the size of pointers. Another approach is to store all the metadata in a *shadow memory*, where the scheme can use the address of a pointer to compute the address of its metadata. For example, HardBound [10] manages pointer bounds in a direct-mapped shadow memory that is logically twice the size as the address space, while Intel MPX [32] manages them in a two-level directory structure to relax the constraint on virtual address space layout. Unfortunately, storing metadata separately impacts performance, as the locality of the metadata with the pointer is lost and the shadow memory address computation can be costly.

Per-pointer bounds can also be extended with additional metadata to enforce other memory-related security properties. WatchdogLite [28], Shakti-MS [9], and ALEXIA [21] can detect *temporal*

[3]The scheme stores and propagates the full pointer bounds along with the pointer value.

Table 1: Comparison between In-Fat Pointer and Related Works on Memory Safety.

(Hardware- or Software-based) defense or software sanitizer [39]	Subject of introduced in-memory metadata for checking	Spatial protection granularity[1]	Lost compatibility[2]	Required feature[3]
Intel MPX [32]	Pointer	Subobject	—	Ⓢ
HardBound [10]	Pointer	Subobject	—	Ⓢ
WatchdogLite [28]	Pointer	Subobject	—	Ⓢ
SoftBound [29]	Pointer	Subobject	—	Ⓢ
CHERI [42]	Pointer	Subobject	ⒷⓈ	Ⓣ
Shakti-MS [9]	Pointer + Object	Subobject	Ⓑ	—
ALEXIA [21]	Pointer + Object	Subobject	Ⓑ	—
BaggyBound[4] [1] [12]	Object/None†	Object	—	Ⓢ/—†
PAriCheck [43]	Object	Object	–	Ⓢ
AddressSanitizer [36]	Memory	Partial	—	Ⓢ
REST [38]	Memory	Partial	—	Ⓣ
Califorms [35]	Memory	Partial	ⒷⓈ	Ⓣ
Prober [26]	None (page permissions only)	Partial	—	—
Low-Fat Pointer [24]	None†	Object	—	—
SMA [8]	None†	Object	—	—
CUP [5]	Object†	Object	—	—
FRAMER [30]	Object†	Object	—	—
AOS [20]	Object†	Object	—	—
EffectiveSan [14]	Object†	Subobject	—	—
ARM MTE [2]	Memory†	Partial	—	Ⓣ
In-Fat Pointer	Object†	Subobject	—	—

† The scheme uses tagged-pointers.

[1] Whether the work can detect spatial memory errors at the bounds of the subobject (Subobject), at the bounds of the object (Object), or the detection is conditional or probabilistic (Partial).

[2] Whether the work increases pointer size and loses binary compatibility (Ⓑ) or requires source code change (Ⓢ)

[3] Whether the work requires a shadow memory(Ⓢ) (can be either software-based or hardware-based) that either grows with the number of metadata subjects or reserves a fixed percentage of memory in the virtual address space, or a hardware-based tagged memory(Ⓣ) [19]

[4] BaggyBound describes (1) a 32-bit system design that uses shadow memory for in-memory bounds table, and (2) a 64-bit system implementation that uses tagged-pointers to store bounds on pointer tags.

memory errors, for example, when a dangling pointer is dereferenced. They use a lock-and-key approach, where the metadata for each pointer also carries a value as "key" that is required to match with the "lock" value from per-object metadata when dereferencing.

2.2 Object-Based and Memory-Based Schemes

Object-based schemes maintain *per-object* metadata in memory, either along with the objects they describe, or with a 1-to-1 mapping from the object [1, 12]. The per-object metadata is usually immutable over the lifetime of the corresponding object and can be efficiently shared among all pointers pointing to anywhere in the same object. However, without per-pointer information, the granularity of protection is limited to the object (or coarser) level. To add per-pointer information, recent work uses pointer tags to lookup in-memory object metadata, and these works are categorized to tagged-pointer schemes described in Section 2.3 below. In-Fat Pointer is able to achieve finer granularity protection by associating a per-pointer *subobject index* with each pointer.

Memory-based schemes associate metadata at a fixed ratio with application memory, and enforce "digested" security policies derived from object allocations, types, or higher-level programming language constructs [35, 36, 38]. AddressSanitizer [36] is a software sanitizer that marks all memory containing live objects as valid with one byte of shadow memory for every eight bytes in the application memory, and catches all the access to invalid memory regions. It also uses a quarantine zone that delays reuse of freed memory locations to assist in catching some temporal memory safety violations. REST [38] and Califorms [35] use hardware-based tagged memory [19] to *blacklist* memory that the application should not access, and the hardware will trap any access to the blacklisted memory rages. Instead of introducing metadata, Prober [26] pads on-heap array allocations with guard pages to catch overflows using the existing page permission checks. While it is possible to use heavy-weight metadata to enforce fine-grained protections (e.g., SDMP [11] assigns pointer-sized tags to each word in memory), most work trades security for low overhead and therefore provides incomplete protection [26, 35, 36, 38].

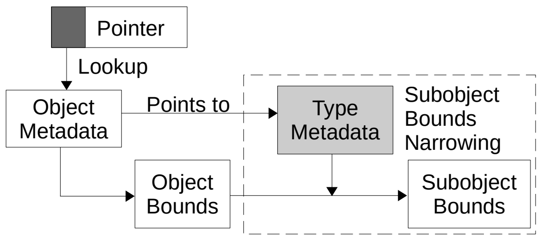


Figure 1: Pointer Bounds Retrieval in Previous Tagged-pointer Schemes using Object Metadata. The subobject bounds narrowing is unique to EffectiveSan [14].

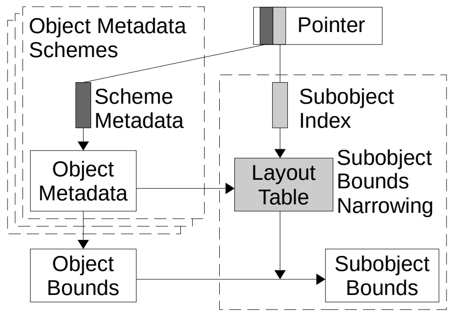


Figure 2: Pointer Bounds Retrieval in In-Fat Pointer

2.3 Tagged-Pointer Schemes

The top bits of a 64-bit pointer are often not used in 64-bit architectures due to physical limitations in addressable memory. These unused bits can thus be used to store pointer metadata, known as a tag. We refer to any scheme that makes use of pointer tags in its design as a tagged-pointer scheme. Note that it is possible for a software protection scheme to manipulate the virtual address space layout to embed metadata in the *used* virtual address bits [13]—we also consider such schemes as tagged-pointer schemes.

Tagged-pointers open up new possibilities for efficient and binary-compatible memory safety enforcement because (1) they can introduce per-pointer metadata without growing pointer size; and (2) the pointer tag can be propagated along with the pointer value without additional instructions to move them. For example, schemes may encode the pointer bounds directly into the tags without in-memory metadata [8, 13, 24]. Memory-based schemes can also use the tags to express stronger security policies. ARM MTE [2] combines 4 bits of a pointer tag with an in-memory tag to achieve probabilistic detection of spatial and temporal memory errors—objects are assigned with a specific tag upon allocation, and all pointers derived from the object will copy the object tag. Therefore a pointer dereference on an object with a tag that doesn't match the pointer tag is a memory error.

More recent works use the pointer tag to lookup an object metadata in memory and retrieve the object bounds from the metadata when a bounds check is needed, as shown in Figure 1. CUP [5] stores all object bounds in a metadata table, and embed the table index into the pointer tag. FRAMER [30] is a generic object metadata scheme that uses 16-bit pointer tags to lookup an object metadata near the object. AOS [20] maintains all heap object bounds in a hash table, and the ARM Pointer Authentication Codes [2] on pointers are used as keys to access the hash table. EffectiveSan [14] uses the software Low-Fat Pointer scheme [13] to lookup the object metadata prepended to each object. However, these works need all pointer tag bits for object metadata lookup, therefore they cannot store mutable per-pointer metadata in the tags and can thus only provide spatial safety at an object bound granularity. The only exception is EffectiveSan [14], which can narrow the bounds to the current subobject by using in-memory type metadata and taking the *use type* of the pointer and the address offset as inputs.

Inspired by the approach of EffectiveSan, In-Fat Pointer also uses a two-stage process for bounds retrieval that involves an object metadata lookup and a subobject narrowing stage, but is able to achieve more efficient subobject narrowing through the use of pointer-tag bits freed up by the use of multiple object-specific metadata lookup schemes. In addition, EffectiveSan depends on accurate type information when retrieving bounds to perform narrowing, so the narrowing is not possible for a `void` pointer. In-Fat Pointer's mechanism does not rely on types during bounds retrieval and thus does not suffer from this limitation.

3 IN-FAT POINTER'S APPROACH

The goal of In-Fat Pointer (short for "Internal Fat Pointer") is to provide hardware-assisted fine-grained spatial memory protection on par with fat pointers without either compromising compatibility with legacy code, as inline fat pointers would (e.g., CHERI [42]), or suffering high overhead from per-pointer metadata, as shadow-space-based fat pointers would (e.g., Intel MPX [32]). To do this, In-Fat Pointer uses a novel pointer bounds retrieval mechanism shown in Figure 2, which uses tagged-pointers to accelerate lookups of in-memory metadata and narrows the object bounds to the most accurate subobject bounds. First, In-Fat Pointer uses three separate object metadata schemes that optimize the tagged pointer space usage so that additional bits can be used to track the dynamic information required for precise subobject bounds checking. Second, In-Fat Pointer keeps track of the currently pointed subobject by maintaining the *subobject index* field on pointer tags so that the subobject bounds for precise checking can be computed by narrowing the bounds from the object metadata. The retrieved bounds are saved in registers and used for bounds checking as traditional fat pointers. In-Fat Pointer is designed for a 64-bit architecture with at least 16 bits available for pointer tags.

Protection Scope and Guarantees. Due to the use of pointer tags as metadata storage, In-Fat Pointer assumes that the application will preserve those pointer tag bits, therefore it does not support applications that modify these bits.

For applications that use pointers as a black box, In-Fat Pointer guarantees that all spatial memory errors will generate exceptions when application source code is available for instrumentation. In-Fat Pointer requires protected binaries to have been compiled with additional instrumentation. For applications that link with legacy, uninstrumented binary libraries, In-Fat Pointer provides no guarantee on overflow of objects allocated in the legacy code, or spatial errors that occur in the legacy code.

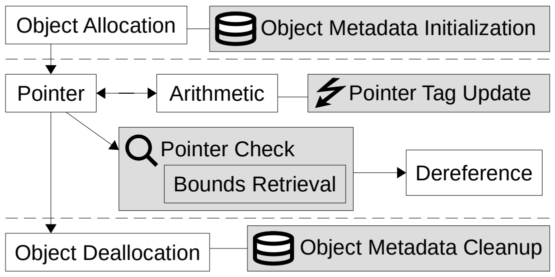


Figure 3: Operations Instrumented by In-Fat Pointer. In-Fat Pointer introduce new instructions for hardware-assisted pointer checks, bounds retrieval (`promote`), and pointer tag updates.

In-Fat Pointer supports pointer casting with `void` pointers. In the presence of incorrect casts on a pointer, In-Fat Pointer may not be able to correctly determine which subobject is being addressed. As a result, In-Fat Pointer guarantees the absence of spatial errors only at the level of the object bounds if a pointer does not have the correct type during bounds narrowing. Otherwise, the same guarantee is provided at the level of the subobject bounds.

In-Fat Pointer supports multi-threaded programs. The introduced object metadata is read-only after initialized during object allocation, therefore these metadata can be safely accessed from multiple threads. However, In-Fat Pointer cannot detect temporal memory errors (i.e., use-after-free) beyond those that invalidate object metadata.

3.1 Overview

In-Fat Pointer uses compiler instrumentation to protect the application code, and introduces new hardware instructions to accelerate commonly instrumented operations. Figure 3 shows the high-level compiler instrumentation performed by In-Fat Pointer. The modified compiler first identifies all pointers whose safety cannot be statically determined and instruments these for runtime checking. In addition, the compiler also adds hardware provided `promote` instructions to retrieve the bounds when they are needed. The `promote` instruction implements the entire object metadata lookup and subobject bounds narrowing procedure shown in Figure 2. To support the bounds retrieval, the compiler instruments object allocation and deallocation to initialize and clean up the object metadata. A modified dynamic memory allocator also assists in instrumenting heap-allocated objects. Finally, the compiler instruments pointer arithmetic with pointer tag update to ensure correct pointer tags for bounds retrieval. In addition to bounds retrieval, the hardware also provides new instructions for bounds checking and pointer tag update that help reduce instruction and performance overhead.

To illustrate, consider the code example shown in Listing 2. `IFP`-prefixed functions represent high-level instrumented code fragments. Inside function `main()`, the compiler will instrument `boo` because its use cannot be statically determined to be safe—a pointer derived from `boo` (`&boo.value`) is saved to a global variable `gv_ptr` and used in another function. Therefore, the compiler will (1) instrument the allocation of `boo` to create object metadata for it; and

```
1  struct Boo {
2    int value;
3    int dummy;
4  };
5
6  // layout table generation
7  const uint64_t __IFP_LT_...[] = {...};
8
9  struct Boo* gv_ptr;
10 void foo();
11
12 int main(void) {
13   struct Boo boo;
14   // object metadata init and pointer tag setup
15   struct Boo *ptr = IFP_Register(
16     /* base addr, size */&boo, sizeof(boo),
17     /* layout table    */__IFP_LT_...
18   );
19   // gv_ptr = &boo.value;
20   // pointer arithmetic with tag update
21   gv_ptr = IFP_GEP(ptr, offsetof(struct Boo, value), ...);
22   foo();
23   IFP_Deregister(ptr);// object metadata cleanup
24   return 0;
25 }
26
27 void foo() {
28   // *gv_ptr = 1;
29   // pointer checking involving bounds retrieval
30   int *ptr = IFP_Check(gv_ptr, sizeof(int));
31   *ptr = 1;
32 }
```

Listing 2: Code Example of In-Fat Pointer Instrumentation.

(2) return a pointer to `boo` that has the correct pointer tag. This instrumentation is represented by `IFP_Register()` at line 15, whose return value is used to initialize `ptr`. Because the type information (`struct Boo`) is available, the instrumentation to object `boo` also triggers the generation of the in-memory type metadata (which we call *layout table*) for `struct Boo` at the compile time, and the information of the layout table is also included in the object metadata. The object metadata will be cleared by `IFP_Deregister(ptr)` at line 23 when `boo` goes out of scope. To ensure that the tag on the pointer `&boo.value` assigned to `gv_ptr` is up-to-date, the compiler replaces the pointer arithmetic of `&boo.value` with `IFP_GEP()` at line 21 and uses its result as the assigned value to `gv_ptr`. The hardware provides specialized instructions to update the pointer tag along with the address computation. Finally, inside function `foo()`, the compiler cannot guarantee that `gv_ptr` is safe to dereference, therefore the pointer is checked with `IFP_Check()` at line 30, which involves a `promote` that retrieve the bounds and a subsequent bounds check. In-Fat Pointer uses *poison bits* on pointer tag to mark the validity of a pointer, and any check failure is indicated by setting the poison bits on the output pointer (also referred to as *poisoning* the pointer). The poison bits indicate whether a pointer is valid or not, and instructions that access memory check these bits. A memory fetch using a poisoned pointer generates an exception. Therefore, the returned pointer from `IFP_Check()` is dereferenced directly. We discuss poison bits in greater detail in Section 3.2. Whenever a pointer needs checking, the compiler will insert a new `promote` instruction is used to *recompute* the pointer bounds from available metadata, and then the bounds can be used for later checking.

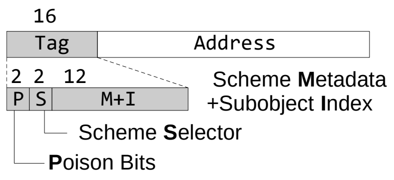


Figure 4: Pointer Tag Bits Decomposition. The numbers above each field represent bit widths of the fields.

To facilitate reuse of retrieved pointer bounds and reduce the number of promotes being executed, In-Fat Pointer introduces *In-Fat Pointer Registers* (IFPRs), which are a set of logical fat-pointer registers that hold bounds information corresponding to pointers stored in general-purpose registers. All new instructions introduced by In-Fat Pointer use IFPRs as storage for pointer bounds; for example, the `promote` instruction takes a 64-bit pointer as an operand and produces an IFPR output (i.e., "promotes" a 64-bit pointer to a fat pointer), and `IFP_Check()` checks a pointer against the bounds stored in the corresponding IFPR. Thus, In-Fat Pointer conceptually uses fat pointers under-the-hood but maintains binary compatibility as these fat pointers are never visible to the application. As a result, the fat pointers are "internal" to our design—hence the name "Internal Fat Pointer".

3.2 Promotion and Pointer Tag Fields

In-Fat Pointer embeds multiple metadata fields on the pointer tags to store necessary metadata for `promote` operations, as shown in Figure 4. The top 16 bits of each pointer are used as tags. The lowest 12 tag bits are used for object metadata lookup and subobject bounds narrowing, and these two fields are referred to as scheme metadata and subobject index. As mentioned in the introduction and shown in Figure 2, In-Fat Pointer introduces multiple object metadata schemes for object metadata lookup. Each metadata scheme encodes distinct metadata on the pointer tag and in the object metadata, and they do not use the same width for scheme metadata. To distinguish object metadata schemes that the pointer (and therefore the pointed object) is using, each pointer tag includes two bits as *scheme selector* to indicate the object metadata scheme in use. The bit pattern that conforms to the canonical addressing (i.e., `00` for pointers in user-level code) are reserved for *legacy pointers*, which are pointers that carry no metadata. These pointers are created by legacy code or used for statically-safe objects in instrumented code. The top two bits of each tag are poison bits that encode whether the pointer is in one of three states: (1) valid, meaning that it points within the bounds; (2) invalid, meaning that the pointer has encountered irrecoverable errors (e.g., invalid object metadata, or indexing into a struct after a failed bounds check) and cannot be dereferenced again; and (3) out-of-bounds but recoverable, which is any other state, notably off-by-one[4]. All standard `load` and `store` instructions check the poison bits and will trap if the poison bit state is anything other than valid, allowing In-Fat Pointer to provide partial protection for legacy code. Poison bits are updated each time an operation changes the value of the underlying pointer, or when a bounds check fails.

[4]C legally permits pointers to points one byte beyond the object's upper bound

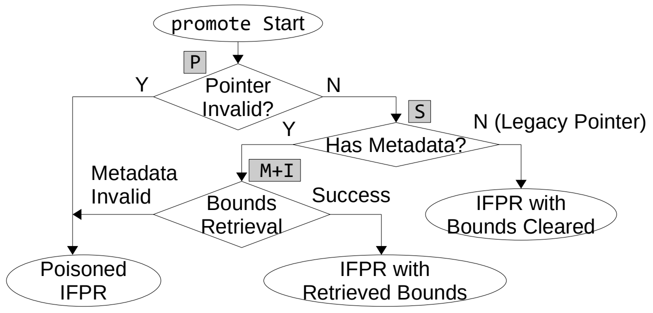


Figure 5: Overview of Promote Operation. Grey boxes represent pointer tag fields in Figure 4 involved in the `promote` operation. The bounds retrieval is presented in Figure 2.

In-Fat Pointer introduces a `promote` instruction to drive the bounds retrieval procedure, as shown in Figure 5. First, `promote` will complete without retrieving any bounds if the poison bits on the pointer tag indicate the pointer is invalid. As we shall see, some object metadata schemes depend on the pointer value to perform a metadata lookup, therefore retrieving metadata with an invalid pointer value may generate an unexpected exception or introduce false positives if the `promote`d pointer is never dereferenced. If the pointer is not invalid, In-Fat Pointer checks the scheme selector field to see if the pointer contains metadata—legacy pointers will have bounds retrieval bypassed, and they will have the bounds cleared and are not subject to bounds checking. Finally, the hardware starts the bounds retrieval shown in Figure 2. It first uses the scheme selector bits from the pointer tag to dispatch the object metadata lookup and object bounds retrieval to corresponding logic. If the object metadata includes layout table information and the pointer is pointing to a subobject (indicated by a non-zero subobject index), then subobject bounds narrowing is performed after the object bounds are retrieved. In this procedure, the `promote` hardware will fetch metadata from memory, and any generated exception (e.g., page faults) will stop the bounds retrieval and the exception is reported as generated from the `promote` instruction. If any metadata is successfully fetched but found invalid, the bounds retrieval is terminated and the output IFPR will be poisoned.

We now present the object metadata schemes and the object metadata lookup in Section 3.3 below. The subobject bounds narrowing is described in Section 3.4.

3.3 Object Metadata Schemes

In-Fat Pointer introduces three complementary metadata schemes, each of which is designed to suit a category of objects. By having separate schemes, In-Fat Pointer can make assumptions on the objects each scheme would serve and is able to make better use of the limited pointer tag bits to reduce the overhead of lookup. Table 2 compares the three schemes on the imposed constraints and the intended use scenario.

Each lookup scheme retrieves object metadata, which contains (1) the base address and size of the object for bounds checking, (2) a pointer to a layout table which describes the size and placement of subobjects for pointer bounds narrowing, and optionally (3) a MAC

Table 2: Object Metadata Schemes Comparison.

	Cons[1]	Use scenario
Local Offset Scheme	`- S -`	Small Objects, Local Variables
Subheap Scheme	`B - -`	Heap-allocated Objects
Global Table Scheme	`- - C`	Global Arrays, Fallback

[1] Whether the scheme impose constraints on a metric:
- The scheme require control on object base address (`B`)
- The scheme has a limit on maximum object size (`S`)
- The scheme has a limit on number of objects supported (`C`)

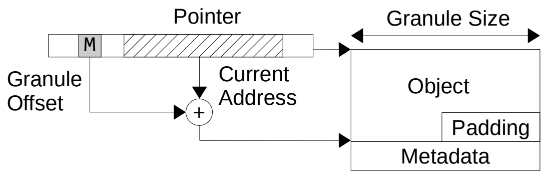


Figure 6: Local Offset Scheme

to protect the integrity of the metadata. Because the metadata is stored in memory and may be vulnerable to modification by legacy code or due to temporal memory errors, the MAC, which is checked during `promote`, is an additional security measure to detect tampering of metadata. In our prototype, the object metadata in each scheme uses different encodings to omit redundant information. Storing the layout table as a pointer allows our prototype to store one layout able for all objects of the same type as they will all have the same layout. We now describe the three metadata schemes.

3.3.1 *Local Offset Scheme.* The local offset scheme is designed for lightweight compiler instrumentation on small objects, especially local variables. Figure 6 provides an illustration of the local offset scheme. The object metadata is appended to each object, and both the object and metadata base address is aligned to an implementation-defined *granule*. The pointer tag contains an offset field, which represents the offset from the current address to metadata in granules. Thus, the `promote` operation retrieves the metadata by adding the address with granule offset and retrieving the metadata. When the address is not granule-aligned, the lower bits are truncated. The local offset scheme also contains a subobject index field in the tag. When pointer arithmetic moves the pointer to a subobject, the instrumented instructions will update both the granule offset and the subobject index. Whereas it is possible to put the metadata at the beginning of an object, placing it at the end means that pointers are usable by legacy code as they still point to the object as opposed to the metadata. Because the metadata address is known from the granule offset, knowing the size is sufficient to derive the object base address. In our prototype, the local offset scheme's object metadata are 128 bits divided into 16 bits for the object size, a 64-bit pointer to the layout table for the object type, and a 48-bit MAC.

The advantage of the local offset scheme is that it has no constraints on the object placement, and is thus suitable for global, heap-allocated, and stack-allocated objects. The main disadvantage is that it places a limit on the size of the objects it can handle, which

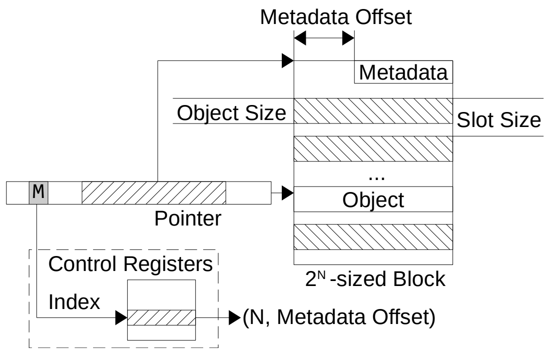


Figure 7: Subheap Scheme. The dashed box represents an implementation defined function that maps assigned tag bits to memory block size and metadata offset.

is constrained by the size of the granule and the offset field in the tag. A larger granule will result in fragmentation, while a smaller granule results in a smaller maximum object size. The tag bits are shared between the object field and the subobject index, so while increasing the object field allows larger objects, it decreases the number of subobjects that can be supported. In our prototype, we use 6 of the 12 bits for the offset, the other 6 bits for the subobject index, and a granule of 16 bytes. Therefore, the implementation can support objects up to $(2^6 - 1) \times 16 = 1008$ bytes in size and object types with at most $2^6 = 64$ elements in the layout table. We expect that most of the objects are smaller than the imposed size limit [7, 45].

3.3.2 *Subheap Scheme.* The subheap scheme intends to be implemented by dynamic memory allocators that group objects with the same sizes or types (e.g., Slab allocator [4]). As shown from Figure 7, the subheap scheme places objects inside power-of-2-sized and aligned memory blocks, where all objects inside the same block share the same copy of metadata in the memory block. The memory allocator will guarantee that only objects having the same size and type (and therefore identical metadata) would be placed inside the same block. Each block contains an array of slots where each slot stores a single object, and the slot has a size constrained to be efficient for hardware to perform division (e.g., power of two or fixed integer multiple of power of two). Besides the layout table information, the common metadata in each block encodes the layout of the slot array, including the start and end offset of the slot array, the size of a slot, and the object size. In the current implementation, the size of the common metadata in each block is 32 bytes. To locate the metadata given a pointer to one of the objects inside the block, the hardware still needs to know the block size and the offset from the block base address to the common metadata base address. In our prototype, we dedicate 16 control registers to store this information. The pointer tag bits in the subheap scheme thus only need to store an index of the control register being used. Thus our prototype uses 4 bits to select the control register, and the remaining 8 bits are used to store a subobject index.

Because the subheap scheme constrains objects to be allocated in the same subheap as other objects of the same size, it is not appropriate for stack-allocated objects. However, unlike the local

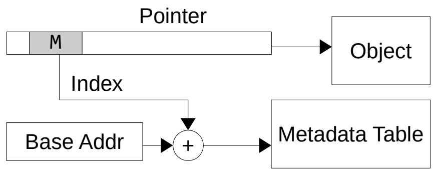


Figure 8: Global Table Scheme

offset scheme, it has no constraints on object size and can support a larger number of subobjects.

3.3.3 Global Table Scheme. The global table scheme is designed to handle objects that cannot use the previous two metadata schemes, for example, global variables that are too large to use the local offset scheme. As shown in Figure 8, it stores all object metadata inside a global table whose base address is stored in a control register. Whenever an object needs to be registered, an empty row in the table is chosen to store the metadata, and the table index is stored in the pointer tag. When the pointer bounds are requested, the hardware simply indexes into the table to retrieve the metadata.

The number of objects that can use the global table scheme is limited by the size of the table, which is limited by the size of the index stored in the pointer tag. In our prototype, we allocate all 12 bits for the index, and thus objects using the global table scheme cannot narrow pointer bounds in `promote`. The size of each row in the global table is 16 bytes.

3.4 Layout Table

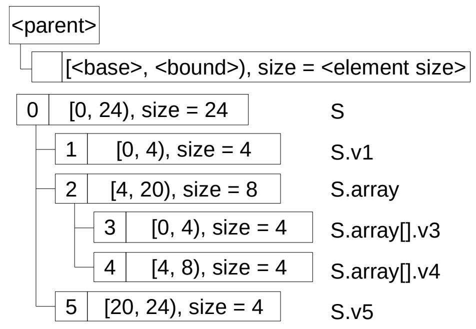
In-Fat Pointer uses a structure called a layout table to encode the relationship between nested objects so that the correct bounds for subobjects can be calculated during bounds-narrowing operations. Consider the example given in Figure 9. The code example shows `struct S`, which contains several elements as well as an array of nested `structs`. Each (sub-)object is given a number (i.e., 0 for `struct S`, 1 for `S.v1`, etc...), which indicates the element that object or subobject occupies in the layout table. Figure 9b illustrates the structure as a logical tree, which is flattened and stored as an array of entries in the layout table. Each element contains the tuple $\{parent, base, bound, size\}$, where the base and bound is the bounds of the subobject represented as offsets from the base address of the parent subobject. For example, for `S.array`, which corresponds to element #2, *parent* would be pointing to element #0, *base* would be the offset of the array field from the base of element #0, `struct S`, *bound* would be the upper bound of the `S.array`, and *size* would be the size of each element in the array (i.e., `sizeof(struct NestedTy)`). Note that the number of elements in `S.array` is not explicitly recorded, but can be computed by evaluating $\frac{bound-base}{size}$.

The compiler instrumentation will update the subobject index to follow changes of the currently pointed subobject. When a pointer is set to point to a subobject, the bounds retrieved by `promote` is narrowed by incrementing the pointer's subobject index to point to the appropriate layout table element, thus updating the lower base and upper bound the pointer can legally point to.

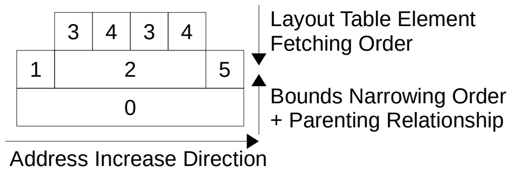
The layout table design optimizes subobject bounds narrowing for common struct nesting. If a type hierarchy only contains struct members or arrays of elementary type, then it can be flattened such

```
struct S {      // 0: struct S
  int v1;       // 1: S.v1
  struct NestedTy {
    int v3;     // 3: S.array[].v3
    int v4;     // 4: S.array[].v4
  } array[2];   // 2: S.array
  int v5;       // 5: S.v5
};
```

(a) Code Example


<parent>
[<base>, <bound>), size = <element size>
0 [0, 24), size = 24 S
1 [0, 4), size = 4 S.v1
2 [4, 20), size = 8 S.array
3 [0, 4), size = 4 S.array[].v3
4 [4, 8), size = 4 S.array[].v4
5 [20, 24), size = 4 S.v5

(b) Generated Layout Table. The top part shows the fields in each layout table element.


3 4 3 4
1 2 5
0
Layout Table Element Fetching Order
Bounds Narrowing Order + Parenting Relationship
Address Increase Direction

(c) Memory View and Operation Orders

Figure 9: Layout Table Example

that all subobject bounds can be computed by indexing into the layout table. Consider the element for `S.v1` and `S.v5` (element 1 and 5) from Figure 9; their bounds are represented as offsets from `struct S` (element 0), which corresponds to the object bounds. Therefore, when narrowing the bounds of a pointer to element `S.v1`, the subobject bounds can be computed by adding the offsets retrieved from the layout table element to the object base address.

For array-of-struct nesting, the subobjects bounds under the nesting are represented as offsets from the parent, which does not correspond to the object bounds. Therefore, the hardware needs to compute the bounds for the parent (sub-)object before child subobjects. In this case, the subobject bounds narrowing is essentially a recursive procedure: to solve the bounds of the current subobject given the subobject index, the hardware will (1) fetch the layout table element to identify the parent element, (2) recursively solve the parent bounds, and (3) narrow the parent bounds according to the fetched element to get the current subobject bounds, as shown in Figure 9c. The base case of recursion is element 0, whose bounds are the object bounds and are known when the narrowing starts. If a pointer to `S.array[1].v3` (which corresponds to element 3) is `promoted`, for example, the hardware will first fetch the layout table elements for `S.array[].v3` (element 3) and then its parent, which is `S.array[]` (element 2). Because the parent of `S.array[]` is `struct S` (element 0), which is the base case, the bounds are the object bounds. Then, the hardware narrows the bounds to `S.array[]` first, compute the base address of the array element for the subobject (`S.array[1]`), and finally add the offset from element 3 to the base address of `S.array[1]` to obtain the bounds for `S.array[1].v3`. While the subobject bounds narrowing in this scenario can be costly, we expect this cost to be rarely paid because pointers to subobjects under array-of-struct nesting tend to be derived from another pointer to the parent (sub-)object, and the derived pointer can obtain the bounds statically without using `promote` instruction in this case. The compiler instrumentation can hoist `promotes` so that only pointers not derived from another pointer (e.g., just loaded from memory) need `promote` to obtain the bounds, therefore it is rare to `promote` pointers to subobjects under deep array-of-struct nesting.

The design of the layout table and subobject index has a number of benefits. First, the subobject index does not need to be updated when performing pointer arithmetic on an array, for example, a `for` loop that traverses `S.array` using pointers. Because all array elements are represented by the single layout table element, no instruction is needed to update the subobject index in such loops, which helps reduce the instrumentation overhead. Second, the layout table is efficient for common subobject bounds narrowing because of the simple indexing with the subobject index from the pointer tags. Finally, the layout table is memory-efficient because all objects of the same type can share a single table, as mentioned earlier.

4 IMPLEMENTATION

In this section, we present the prototype implementation on 64-bit RISC-V architecture. We implemented the hardware prototype of In-Fat Pointer on CVA6 [44], an open-source 64-bit RISC-V processor core with a single-issue in-order pipeline implementing RV64IMAC features. We modified the RISC-V Linux port from the PULP platform [33] (based on Linux kernel 5.1.0-rc7) for the operating system support. The compiler instrumentation is implemented in Clang/LLVM [25] 10.0.0. We also link in a runtime library to initialize the additional states and support the metadata schemes.

4.1 ISA Extension and Hardware Support

Table 3 lists the essential instructions introduced by In-Fat Pointer. Our In-Fat Pointer implementation introduces one bounds register for each of the 32 general-purpose registers (GPRs), and each (GPR and bounds) register pair is treated as a logical IFPR. Each bounds register is 96 bits (48×2 bits) in size, and can be loaded or stored with `ldbnd` or `stbnd` instructions. The `promote` instruction encapsulates the bounds retrieval operations, and the compiler will instrument them when it cannot statically determine the bounds for a pointer. If the object a pointer points to is known, or the bounds of a pointer need to be narrowed to a specific size, the compiler will instrument `ifpbnd` instructions to set the bounds to the known size. When the pointer needs to drop the associated bounds, for example, when the pointer will be stored to heap memory, In-Fat Pointer instruments `ifpextract` to "demote" the IFPR into GPR,

Table 3: Core Instructions from In-Fat Pointer

Mnemonic	Description
`promote`	pointer bounds retrieval
`ifpmac`	MAC computation
`ldbnd`	load bounds from memory
`stbnd`	store bounds to memory
`ifpbnd`	create pointer bounds with given size
`ifpadd`	address computation and tag update
`ifpidx`	subobject index update
`ifpchk`	(bounds) access size check
`ifpextract`†	extract fields from IFPR / demote
`ifpmd`†	pointer tags manipulation

† Multiple variants of the instruction exists.

which is essentially a truncation but will also update the poison bits if the pointer is (wildly) out-of-bounds and the exact bounds cannot be retrieved. The `ifpadd` instruction is used to implement address computations, for example when indexing into an array or a struct. If the code is indexing into a struct, besides the address computation, the compiler will also instrument `ifpidx` to update the subobject index on the pointer tag, and instrument `ifpbnd` to narrow the pointer bounds if needed. In-Fat Pointer implements bounds checking as an *access size* check, which checks both the current address against the lower bound and the address *plus* a size against the upper bound. To reduce the instruction overhead from pointer checks instrumentation, In-Fat Pointer can check a pointer either *explicitly* by instrumenting `ifpchk` instructions, or *implicitly* by using a subset of registers as the address operand of loads and stores. The implicit size checks are explained in Section 4.1.1 below. In addition, `promote` fuses a size check and `ifpadd` will update the poison bits when the address computation result is out of bounds. To reduce the instruction overhead from object allocation instrumentation, In-Fat Pointer provides `ifpmac` for MAC generation and `ifpmd` for pointer tags setup when instrumenting object allocations.

The prototype implementation introduces a new execution unit, *IFP unit*, to the execute stage of the processor pipeline. The IFP unit implements the `promote` and `ifpmac` instructions. The load-store unit from CVA6 is modified to (1) implement `ldbnd` and `stbnd` instructions, (2) perform implicit size checks and poison bits checks on address operands, and (3) serve metadata load requests from the IFP unit. The remaining instructions in Table 3 are implemented in the integer ALU and take a single cycle to execute.

4.1.1 Implicit Checking. In-Fat Pointer implementation selects a subset of IFPRs (referred to as *bounds-checked* IFPRs) to perform *implicit* bounds checking to reduce the instruction overhead. When a RISC-V load or store takes a GPR from the selected IFPRs as the address operand, the instruction issue logic will read the bounds associated with the pointer, and the load-store unit will check the bounds for the dereference. In this way, when the application code contains a loop that iterates over an array, In-Fat Pointer can incur no overhead on bounds checking by allocating the pointer loop variable into a bounds-checked IFPR. However, to prevent uninstrumented callee from accidentally picking up bounds from an instrumented caller when dereferencing pointers in callee-saved registers, the implicit bounds-checking should only be enabled when the bounds are caller-saved. Therefore, the In-Fat Pointer implementation apply implicit bounds checking for caller-saved registers. Bounds checking on other registers still require `ifpchk` instructions.

4.1.2 Calling Convention. In-Fat Pointer implementation extends the RISC-V calling convention to apply the same caller-callee saving and argument passing rules on GPRs to the bounds registers. Each function will save and restore all clobbered callee-saved registers, including both the bounds registers and the GPRs. In addition, if pointers are passed as function arguments or return values, the bounds should be passed in the corresponding bounds registers. This allows In-Fat Pointer to pass pointer bounds across functions more efficiently, without performing full promotion for all pointer arguments and return values. This property is also significant for our implementation to provide subobject granularity protection when there is no layout table for the object, because the bounds can be inherited from previous bounds narrowing operations. However, without additional measures, the pointer bounds may not match with the pointer value when there are calls between uninstrumented and instrumented code. For example, when the calling convention uses a register for both a call argument and the return value, an instrumented caller may obtain wrong bounds of a pointer returned from the uninstrumented callee; the callee will not update the bounds and the resulting bounds likely come from a function argument that the caller passed in. To address this problem, In-Fat Pointer introduces implicit *bounds clearing*; when a GPR involved in an argument or return value passing is modified by an existing instruction from RISC-V (i.e., not introduced by this work), the associated bounds register is automatically cleared by the hardware. Using this approach, the legacy binary either leaves the register value intact, in which case the bounds register still contains the correct bounds, or modifies the register using an existing instruction, which clears the bounds register. The instrumented code will therefore never obtain invalid bounds.

Because RISC-V has a single standard calling convention, the implementation currently enables implicit bounds checking and clearing on all caller-saved registers. For an architecture with multiple calling conventions, one possible implementation is to introduce a control register to encode which registers to enable implicit bounds checking and clearing. For applications that mix calling conventions in a single process, the selected set can be the union of caller-saved registers; this may result in callee-saved GPRs to pair with caller-saved bounds registers, but the compatibility can be preserved. If a pointer argument is passed in memory, the prototype implementation will demote the pointer and drop the bounds before saving the pointer to memory, and promote the loaded pointer after loading it back. We leave binary-compatible bounds-passing for in-memory arguments as future work.

4.2 Software Support

The In-Fat Pointer prototype implementation modifies the software stack from the operating system to the applications. First of all, the OS kernel is modified to save and restore additional registers when entering and leaving kernel mode, as well as ignoring pointer tags (but not poison bits) when checking pointers from user space. Next, the implementation introduces a runtime library to (1) provide allocator functions for heap allocations, (2) initialize the In-Fat Pointer environment at application startup, and (3) manage the global metadata table from the global table scheme. Lastly, the modified compiler is used to analyze and instrument the programs.

4.2.1 Dynamic Memory Allocators. To evaluate In-Fat Pointer with dynamic memory allocators that may not support subheap scheme, we implemented two allocators in the runtime library: (1) a wrapped allocator on top of `libc`'s `malloc()` and `free()` which uses the local offset scheme if possible and falls back to the global table scheme; and (2) a subheap allocator implemented as a pool allocator on top of a buddy allocator, which uses the subheap scheme. The wrapped allocator transparently over-allocates memory for objects when using the local offset scheme, and allows us to evaluate the impact on existing allocators that cannot support the subheap scheme. The subheap allocator implementation, however, intends to model state-of-the-art scalable memory allocators [15, 23, 27] modified to support the subheap scheme.

To utilize the provided allocator functions, during compiler instrumentation, all `malloc()`, `free()`, and other allocator calls in the source programs are rewritten to call the counterparts from the runtime library. For `malloc()`-like allocation functions, extra information (e.g., layout table metadata) is passed as additional arguments to the allocator. The runtime library guarantees that the pointers returned from the allocation functions (e.g., `malloc()`) have the tags properly initialized. The prototype implementation uses a compiler option to select which allocator to use for all such allocator function calls. However, we note that it is possible to use both allocators simultaneously and the runtime library can dynamically select allocators and metadata schemes. We leave such exploration as future work.

4.2.2 Local and Global Objects Instrumentation. For local objects, the compiler will instrument metadata setup and cleanup code into the functions they are declared. The compiler will prefer the local offset scheme if the object is within the size limit, and fall back to the global table scheme otherwise. The runtime library provides functions to register and deregister the objects for the global table scheme, and the compiler will instrument calls to these library functions if the global table scheme is used.

For each global objects requiring instrumentation, the compiler will create a "`getptr`" function in the program, and this function is called when the application needs to obtain the pointer with tag. It will initialize the object metadata once in its first invocation, and simply return the pointer in subsequent calls. The compiler will create stub code in source files that only contains the declaration of the object, and create complete code in the source file the object is defined. The compiler will set the linkage so that the `getptr` function from the source file defining the object is prioritized during linking. Therefore, the stub version of `getptr` will only be used when the object is defined in an uninstrumented source file. Although getting a pointer to the global variable would involve a function call, it is only needed for code that makes the pointer escape, e.g., storing the address of the object to memory. Referencing global variables by name, for example, is not affected by the instrumentation.

5 EVALUATION

We evaluate In-Fat Pointer (1) functionally by testing whether In-Fat Pointer can successfully catch spatial memory errors at runtime, (2) by measuring the performance and memory overhead of benchmarks and real workloads, and (3) by estimating hardware area overhead from our prototype implementation.

We used Xilinx Vivado 2018.2 to synthesize our modified CVA6 on a Digilent Genesys 2 FPGA development board, which has a Kintex-7 XC7K325T-2FFG900C FPGA chip and 1GB of DDR3 RAM. The processor runs at a fixed frequency of 50MHz. We run all experiments with `/proc/sys/vm/overcommit_memory` set to 1 to accommodate the subheap allocator implementation.

5.1 Functional Evaluation

To evaluate the correctness of In-Fat Pointer, we ran the NIST Juliet test suite 1.3 for C/C++ [31]. There are 64,099 total test cases in the Juliet test suite, where each test case is a program with both good and vulnerable code fragments, and the `main()` function calls the good code first, then the vulnerable code. We chose programs under the category of buffer overflow (stack-based and heap-based), underwrite, overread, and underread, which features out-of-bounds memory access that In-Fat Pointer intends to catch. Among the total of 16,572 test cases in the selected categories, we excluded all 6,484 C++ test cases because our prototype currently does not support C++. For the remaining 10,088 test cases, we excluded those that have external dependencies (e.g., requiring network connections, files, or input from `stdin`), and those that only run on Windows, leaving 9,328 test cases. We found that at least 182 test cases only exhibit spatial errors when a randomly generated value falls into specific ranges in the first round of experiments, so we modified the random number generation function to generate specific values according to an environment variable so that these programs exhibit spatial errors at least once when multiple runs with different environment variable values are used. After the adjustment, among all the 9,328 test cases, we found that 2,672 of them either have the spatial error triggered in uninstrumented `libc`, or do not involve spatial errors on our test environment because they either require special (non-zero) initial value in uninitialized memory or are only vulnerable on a 32-bit system. Among the rest 6,656 test cases, there are 1,084 of them have their spatial error optimized away by the compiler, leaving 5,572 valid test cases for In-Fat Pointer. We ran all 5,572 vulnerable and non-vulnerable test cases on our In-Fat Pointer prototype and found that In-Fat Pointer successfully detected all vulnerabilities while passing all non-vulnerable cases. While all intra-object overflow test cases have the overflow optimized away by the compiler and therefore excluded from the evaluation, the result validates our prototype for object-granularity protection, which increases our confidence of correctly implemented subobject-granularity protection.

5.2 Application Evaluation

To show that In-Fat Pointer is a practical approach for spatial memory safety defense, we run all programs from Olden [6], four programs (anagram, ft, ks, and yacr2) from PtrDist [3], and the following selected programs: (1) bzip2 [37] 1.0.8 compressing its own source code tarball, (2) 458.sjeng from SPEC2006 benchmark [17] with reduced search depth and runtime, (3) CoreMark [16], and (4) WolfCrypt's Diffie–Hellman benchmark [41]. These four programs are selected because they cover common application domains and can run on our prototype with limited resources and complete in a reasonable time. We modified yacr2's file parsing code to embed input data into the program to circumvent an implementation bug. PtrDist's bc is excluded because In-Fat Pointer detects the over-read from `glibc`'s `strlen()` in its instrumented versions[5].

To measure the runtime overhead from the `promote` instruction, we implement a version of In-Fat Pointer we call *no-promote*. No-promote is identical to the standard In-Fat Pointer prototype except that `promote` treats all pointers as legacy pointers and do not perform any metadata access, therefore `promote` has the same cost as a `nop`. By comparing the performance of no-promote with the standard version of In-Fat Pointer, we can then isolate the performance impact of the `promote` instruction. We note that no-promote has no difference in memory overhead and has essentially identical memory usage as the standard version.

Our experiments show that uninstrumented programs have no observable change in runtime or memory consumption when running on modified Linux or vanilla Linux, so all our analysis below uses uninstrumented programs running on modified Linux as a baseline. All reported overheads are normalized to the baseline results.

5.2.1 *Instrumentation Statistics.*

Table 4 lists the statistics on object instrumentation, `promote`, and the total dynamic instruction counts. The results involving layout table and subobject index usage are collected from the subheap allocator versions of the programs. The local offset scheme is used for all local objects and most of the global objects, except one global object from sjeng and three from bzip2 using the global table scheme. The metadata scheme for each heap-allocated object depends on the used allocator. Therefore, the metadata scheme usage of each program version can be computed from the object instrumentation statistics. The subheap allocator versions have 5% geo-mean dynamic instruction increase and the wrapped allocator versions have a 14% increase. The difference results from the performance difference of the dynamic memory allocator which is explained in Section 5.2.2 below.

In our experiments, In-Fat Pointer provides the same protection granularity as traditional fat pointers in 16 out of 18 programs except for CoreMark and bzip2. Most of the `promote`s only need to retrieve object bounds; among the 18 programs, only CoreMark, bzip2, and health contain `promote` of pointers to subobjects[6] (29%, 50%, and less than 1% of `promote` instructions respectively), and our results indicate that all subobject bounds narrowing in health are successful while all narrowing in CoreMark and bzip2 fail because the object metadata does not contain layout table information, and the bounds are coarsened to the object granularity. We found that CoreMark, bzip2, and wolfcrypt use custom wrapper functions for memory allocation, and both bzip2 and wolfcrypt use function pointers to invoke memory allocation functions. Our compiler instrumentation therefore cannot deduct the allocated type and create

[5]The `strlen()` implementation makes word-sized (64-bit) read for word-aligned strings, including short strings allocated with less than 8 bytes of memory.

[6]These `promote` instructions take a pointer operand with non-zero subobject index.

Table 4: Dynamic Event Counts on Object Instrumentation, Promotion, and Instructions Executed

Benchmark	Global objects[1]		Local objects[1]		Heap objects[1]		Valid `promote`[2]		Dynamic instruction counts		
	# Object	% LT	# Object	% LT	# Object	% LT	# `promote`	% Total	Baseline	Subheap	Wrapped
bh	0		1.24×10^7,	100%	6.15×10^3,	33%	4.89×10^6	(99%)	1.11×10^9	1.21×	1.21×
bisort	0		0		1.31×10^5		1.53×10^7	(55%)	3.38×10^8	1.04×	1.14×
em3d	0		0		1.60×10^4		3.25×10^6	(≈ 100%)	2.38×10^8	1.11×	1.11×
health	0		0		1.73×10^5,	< 1%	2.44×10^7	(94%)	2.02×10^8	1.10×	1.32×
mst	0		0		838,	< 1%	9.15×10^6	(78%)	1.78×10^8	1.12×	1.11×
perimeter	0		0		1.40×10^6		1.95×10^7	(82%)	1.03×10^9	0.80×	1.11×
power	0		0		1.83×10^4,	9%	1.08×10^6	(100%)	1.54×10^9	1.00×	1.00×
treeadd	0		4,	100%	2.10×10^6		2.10×10^6	(50%)	8.22×10^8	0.61×	1.21×
tsp	0		0		1.31×10^5		1.04×10^7	(99%)	3.37×10^8	1.00×	1.09×
voronoi	0		1.67×10^3,	100%	3.28×10^4,	< 1%	4.34×10^6	(44%)	3.29×10^8	1.04×	1.07×
anagram	3,	33%	1,	100%	314,	≈ 100%	2.37×10^5	(41%)	2.01×10^7	1.18×	1.18×
ft	0		1		9.01×10^4,	< 1%	2.27×10^8	(≈ 100%)	1.50×10^9	1.22×	1.28×
ks	4		0		2.01×10^3		6.67×10^8	(79%)	4.20×10^9	1.21×	1.21×
yacr2	0		0		85,	2%	1.97×10^6	(≈ 100%)	2.02×10^8	1.16×	1.16×
wolfcrypt-dh	1,	100%	592,	89%	8.34×10^3		2.51×10^6	(≈ 100%)	3.25×10^9	1.14×	1.13×
sjeng	5		4.69×10^6,	< 1%	4		2.08×10^6	(26%)	5.68×10^9	1.06×	1.06×
coremark	2,	100%	2,	50%	1		2.12×10^7	(98%)	4.53×10^8	1.08×	1.08×
bzip2	4,	25%	9,	33%	11		9.07×10^6	(74%)	8.31×10^9	1.09×	1.09×

[1] Each cell contains (1) the number of objects instrumented, and (2) percentage of object metadata which includes layout table metadata. If no layout table is used, the zero is omitted.

[2] Each cell contains (1) the number of `promote` not bypassing object metadata lookup are executed, and (2) their ratio within all `promote`.

layout tables for these allocations. We leave support for allocation wrappers as future work.

Another observation is that more than 20% of `promote` instructions executed on average have `NULL` or legacy pointers as input. The approach used by In-Fat Pointer can perform better than traditional fat pointers in such scenarios because In-Fat Pointer can bypass the bounds retrieval for legacy or `NULL` pointers while traditional fat pointers cannot. Table 4 lists the statistics on `promote` that access object metadata in the Valid `promote` column. The ratio of `NULL` and legacy pointers operands depend on the program. In bisort, health, perimeter, treeadd, ks, and CoreMark, almost all `promote` bypassing metadata lookup encountered a `NULL` pointer. In voronoi, anagram, sjeng, and bzip2, almost all such `promotes` encounter pointers from legacy code. Among 23% of `promotes` that bypass metadata lookup in mst, 60% encounter a legacy pointer and 40% encounter a `NULL` pointer. In our experiments, these legacy pointers mostly come from the uninstrumented `libc`. For example, the source code of anagram contains multiple calls to `isalpha()` in loops, and each of them compiles to a call to `__ctype_b_loc()` that return a double pointer to a character traits table, a dereference on the double pointer followed by a `promote` on the result pointer, an array indexing that load the trait of the character, and a bitfield test on the trait. The instrumented `promote` will always get legacy pointer operands. We leave better support for `libc` functions and reduction of unnecessary `promotes` for future work.

We also found that almost all valid `promotes` retrieve bounds for pointers to heap-allocated objects; only CoreMark and sjeng has negligible (less than 1%) `promote` for local or global objects. This shows that passing pointer bounds through function arguments and return values is effective at reducing the number of executed `promotes`. The following discussion will also emphasize more on heap-allocated objects than local or global objects.

5.2.2 Performance Overhead. Figure 10 shows the normalized runtime overhead for all programs. The subheap allocator versions incur about 12% geo-mean overhead, and the wrapped allocator versions incur 24% overhead. These overheads are lower than the reported values from software-only FRAMER (223%) and the hardware-based Intel MPX (50%).

First of all, our subheap allocator implementation is more efficient in handling frequent dynamic allocations, especially the allocations of objects with the same size and type, than the allocator from `glibc`. Therefore, the subheap allocator versions of both perimeter and treeadd outperform the baseline because they perform a lot of dynamic allocations as shown in Table 4 and the performance gain from the allocator exceeds the instrumentation overhead.

For the rest of the programs, the largest contributing factor of the overhead are `promote` instructions. Figure 11 shows the counts of new instructions introduced by In-Fat Pointer including `promote`. The instruction counts are normalized against the baseline instruction counts in Table 4. The wrapped allocator versions of health and ft have the highest overhead (81% and 93% respectively), and as shown in the figure, `promotes` make up more than 10% of the instruction overhead introduced by In-Fat Pointer. The baseline of the two programs suffer from cache thrashing; they have approximately one L1 data cache miss in every 8 and 6 instructions executed, respectively. In this case, the metadata sharing in the subheap scheme reduces the memory footprint of the metadata and results in fewer cache misses. The wrapped allocator version

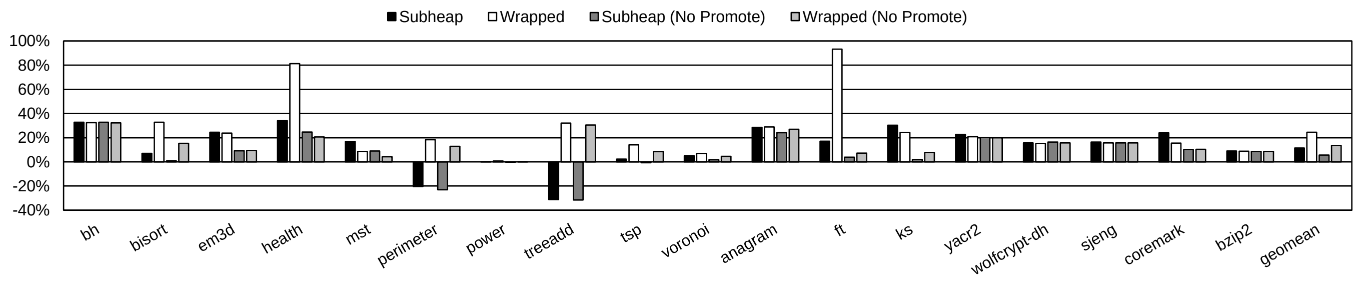


Figure 10: Performance Overhead of All Benchmarks

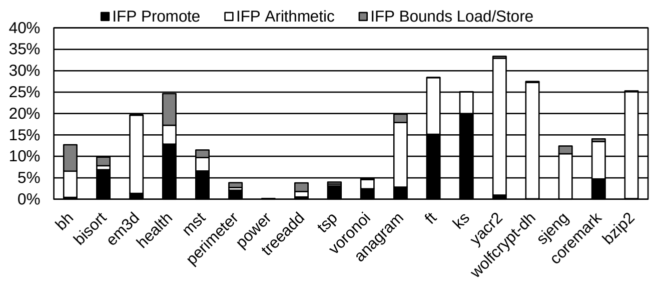


Figure 11: Dynamic Instruction Counts for Instructions from In-Fat Pointer

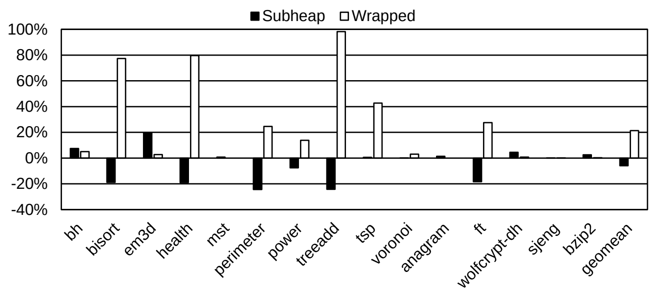


Figure 12: Memory Overhead of Applicable Benchmarks

increases L1 data cache misses by 93% and 96%, while the subheap allocator version increases cache misses by 26% and ≈ 0%, respectively. However, when the metadata can fit in the cache, the subheap scheme is slower than the other two schemes because of the larger metadata and unpipelined metadata fetching from the IFP unit. For example, ks has 17% `promote`s in the dynamic instruction count, and its subheap allocator version is slower than the wrapped allocator version. We also note that C programs written for resource-constrained environments tend to make fewer dynamic memory allocations and therefore cannot benefit from the metadata sharing of the subheap scheme. For example, CoreMark only performs a single dynamic allocation and builds all data structures inside the allocated memory. It has about 5% `promote`s among all instructions executed, and the overhead is 24% for the subheap allocator version and 15% for the wrapped allocator version.

Although `promote` instructions are slow, they contribute to less than 2% of total executed instructions in 10 out of 18 benchmarks. As shown in Figure 11, In-Fat Pointer also instruments many arithmetic instructions that update the pointer tags and bounds or maintain object metadata. We expect future work to use better compiler optimizations including whole-program analysis to optimize away more pointer checks and object instrumentations to reduce the overhead.

5.2.3 Memory Overhead. Figure 12 shows the normalized memory overhead (maximum resident size reported by `time -v`) for both the wrapped allocator version and the subheap allocator version of the programs. Three programs (ks, yacr2, and CoreMark) are excluded in the figure because they use less than 6MB of memory in all versions and `time -v` reports no difference in maximum resident size. The geo-mean overhead from the rest of the programs is −6% for the subheap allocator version and 21% for the wrapped allocator version. The overhead from the wrapped allocator version is comparable with FRAMER [30] and CHERI [42], is much lower than Intel MPX's 1.9× to 2.1× [32].

Comparing with the allocator from `glibc`, the subheap allocator reduces the memory footprint in 6 of the 15 benchmarks. These benchmarks allocate objects individually (i.e., `malloc(sizeof(T))`), therefore the subheap allocator implementation can tightly pack these objects without per-object allocation metadata as the allocator from `glibc` would. On the opposite, `em3d` contains many array allocations for `struct`s (i.e., `malloc(num*sizeof(T))`), and the subheap allocator must allocate arrays with different sizes to different memory blocks, resulting in the highest memory overhead among these benchmarks.

5.2.4 Performance Prediction on ASIC Implementation. We may take several factors into account when trying to extrapolate the performance of our FPGA prototype to an ASIC implementation. First, the CVA6 core that our In-Fat Pointer prototype is based on is a simple single, in-order issue processor with relatively small caches. Second, the clock speed of our FPGA core is much slower than an ASIC compared to the speed of memory. As shown in Figure 11, in some of the programs, `promote` instructions are rare and most of the dynamic instructions introduced by In-Fat Pointer are arithmetic instructions that do not access memory. An out-of-order, superscalar processor may thus be able to hide the costs of the additional In-Fat Pointer instructions better thus having lower overhead than our prototype. However, there are programs that use pointers heavily and more `promote` instructions are executed than other arithmetic instructions introduced by In-Fat Pointer. The

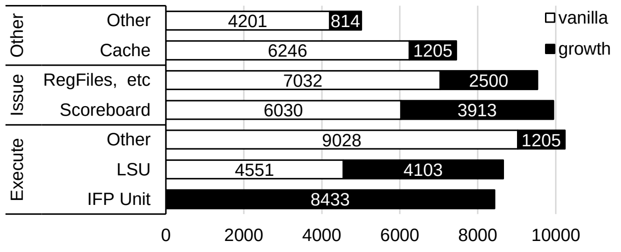


Figure 13: LUT Increase in the Modified Processor

overhead will therefore also be application dependent, as data dependencies (especially on pointer values) may still limit the amount of instruction-level parallelism, and `promote` instructions may increase the load-to-use latency of pointers. We expect the relatively slower memory as compared to processor clock speed to have minimal impact, as In-Fat Pointer does not affect caches, and an ASIC implementation will likely have larger caches.

5.3 Hardware Cost Evaluation

To evaluate the hardware cost of our prototype implementation, we analyzed the resource utilization reports from Vivado. Although the resource utilization on FPGA is not a reliable prediction for chip area in ASIC implementations, it reflects the relative logic complexity of introduced hardware changes and shows possible trade-offs among hardware area cost, memory safety protection features, and performance. In addition, the FPGA resource utilization is relevant for embedded systems using soft-core processors synthesized on FPGAs.

While the vanilla CVA6 processor uses 37,088 LUTs and 21,993 FFs, our modified code uses 59,261 LUTs and 32,545 FFs in total, which is about 60% increase in LUTs and 48% in FFs. Because in almost all hierarchies the design consumes more LUTs than FFs, all the following area analyses use LUTs to approximate the area cost.

Figure 13 shows the LUT usage decomposition at each pipeline stage of CVA6. Among all the pipeline stages, about 62% of the overall utilization increase comes from the execute stage, where the IFP unit and the load-store unit contributes most to the increase (38% and 19% respectively). The usage increase in load-store unit mainly comes from the widened buffers and logic to implement bounds load and store. The 29% usage increase in the issue stage comes from the bounds registers and their operand forwarding logic, and an additional writeback port from the IFP unit. The rest of the changes, including data bandwidth improvement to the data cache, additional control registers and performance counters, and instruction decoding logic incurs less than 10% increase. The usage increase from the IFP unit is 38%. Therefore, if a system using a 64-bit soft-core processor wants to implement IFP support but at the same time limit the area overhead below 30%, the implementation should consider avoiding implementing additional bounds registers and redesign the instruction set, because the bounds registers contribute more LUT usage increase than the IFP unit.

In the IFP unit, the layout table walker uses 3,059 LUTs, which is 36% of LUTs in the IFP unit. In comparison, the logic for all three object metadata schemes uses 2,501 (30%) LUTs. The layout table walker is the most complex component in the processor modification. In the prototype, it has 1,030 lines of SystemVerilog code, while the three object metadata schemes have 676 lines of code in total. Most of the complexity arises from the support for array-of-struct nesting, in which case the layout table walker needs complex state machines and multi-cycle division logic. Therefore, if a soft-core-based system is concerned about performance but wants to reduce area, the IFP implementation may simplify or drop support for layout table and rely on bounds narrowing in application code for fine-grained spatial protection.

6 CONCLUSION

In this paper we present In-Fat Pointer, a hardware-based tagged-pointer scheme that can provide spatial memory safety guarantee at subobject bound granularity without breaking compatibility with legacy code. We show that multiple object metadata schemes can reduce both the demand of pointer tag bits and the cost of object metadata lookup, and the spared tag bits can be used to assist subobject bound recomputation with the help of per-type layout table. We implemented and evaluated In-Fat Pointer on RISC-V architecture.

In the future, In-Fat Pointer can be improved with better compiler analysis and optimization to reduce the performance overhead and support more complex programs with full coverage. The design parameter selection of In-Fat Pointer metadata schemes will also benefit from a better knowledge of application behaviors. We also look towards better metadata scheme designs that can make more efficient use of pointer tag bits and the object metadata.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd, Michael Swift, for their thoughtful and insightful comments. We would also like to thank Tony Liao, Tarek Abdelrahman and Natalie Enright Jerger for their feedback. This work is supported in part by NSERC CRD Grant CRDPJ 541942-19 and funding from Huawei Canada.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact mainly contains the source code for (1) the modified CVA6 processor, (2) the modified Clang/LLVM compiler, (3) the modified Linux kernel, (4) the runtime library for supporting In-Fat Pointer instrumentation, and (5) patches and configuration files for Ariane-SDK and its RISC-V GNU toolchain to generate the bootable Linux image and provide the linker and `glibc` for the Clang/LLVM toolchain. Unfortnately, we could not make the artifact publicly available.

A.2 Artifact Check-List (Meta-Information)

- **Program: Benchmarks used in Section 5.2**
- **Compilation: Modified Clang/LLVM compiler**
- **Run-time environment: Modified RISC-V 64-bit Linux**
- **Hardware: Digilent Genesys 2 FPGA Board**
- **Metrics: Runtime and Memory Overhead reported by `time -v`**
- **Output: Terminal Logs that can be parsed to produce Figure 10 and 12**
- **How much disk space required (approximately)?: 70-120GB**

- **How much time is needed to prepare workflow (approximately)?: 6-12 hours**
- **How much time is needed to complete experiments (approximately)?: 14+ hours**
- **Publicly available?: No**

A.3 Description

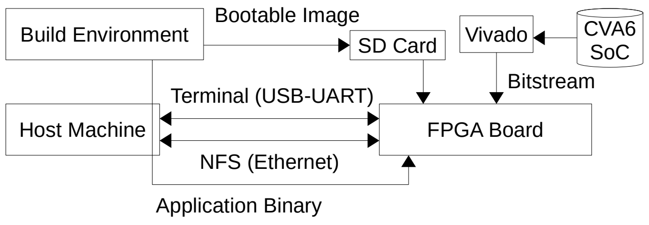


Figure 14: Overview of Prototype Setup

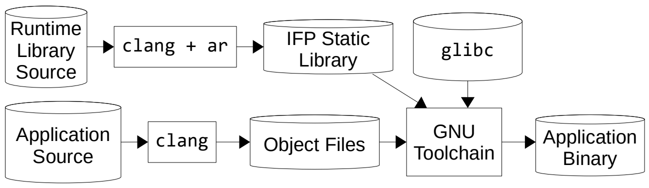


Figure 15: Software Build Procedure

Figure 14 shows the general setup of the prototype implementation. A build environment is configured to prepare the application binaries and the bootable Linux image. The image is written to an SD card for use by the FPGA system. We use the Digilent Genesys 2 FPGA board for the prototype system, and the bitstream is generated from the CVA6 SoC with the modified processor with Xilinx Vivado 2018.2. A host machine is physically connected to the FPGA to communicate with the system. We used the USB UART serial port for the terminal and NFS over Ethernet for file sharing.

Figure 15 shows how the application binaries are built in the build environment. The runtime library is first compiled and archived into a static library, and then linked into all instrumented programs. We use the modified Clang/LLVM compiler to cross-compile all source files to RISC-V object code, including the runtime library. We then use the RISC-V GNU toolchain from the patched Ariane-SDK[7] for linking. To compile and instrument applications with minimal change to the normal build procedure, our implementation includes wrapper scripts as drop-in replacements to `CC` for Makefiles and `configure` scripts.

A.3.1 How to Access. We provided reviewers with a docker image containing the cross-compilation toolchains, and remote access to our machine connected with the FPGA board. This artifact is not publicly available; please contact the authors if access is desired.

[7]The upstream Ariane-SDK is available at https://github.com/pulp-platform/ariane-sdk. The Ariane processor was renamed to CVA6 around the time of publication.

A.3.2 Hardware Dependencies. A Digilent Genesys 2 FPGA board is required to run the experiments. A Linux-capable host machine with at least one USB port and one Ethernet port is needed to communicate with the FPGA system.

A.3.3 Software Dependencies. Xilinx Vivado 2018.2 is needed to synthesize the SoC and generate the FPGA bitstream. Vivado must have a license for synthesizing and implementing on the Xilinx XC7K325T-2FFG900C FPGA device (which is from the Kintex-7 series). Note that Vivado's free WebPACK edition does not contain these licenses.

A.4 Installation

The setup of the build environment and the creation of application binaries and the Linux image consists of the following steps:

- Patch Ariane-SDK and install the RISC-V cross-compilation toolchain
- Patch Linux kernel and build the boot image with the Ariane-SDK
- Build the Clang/LLVM cross-compilation toolchain
- Build the runtime library
- Build the application binaries

We have scripts for most of the steps, including the configuration and building of the applications.

The FPGA bitstream generation procedure of the modified CVA6 is identical to the procedure in upstream CVA6 repository[8]. During experiments, we run the following command on the host machine to open a terminal to the FPGA system after the cables are correctly connected:

```
$ sudo screen -L /dev/ttyUSB0 115200
```

In all evaluations, the execution statistics of the programs are printed to the terminal, and `-L` option of the `screen` command will save the logs to a text file. We use scripts to parse all the terminal logs and generate spreadsheet files.

A.5 Experiment Workflow

The performance evaluation of In-Fat Pointer involves (1) build the applications[9], (2) copy the application binary to the NFS share on the host machine (if the build environment is not on the host machine), (3) run the benchmarks with the inputs in Listing 3, and (4) parse the terminal log file to extract the printed statistics. We have scripts to automate all the steps above (except the second step). `$FILES` points to the directory with the input files. The input files for PtrDist are distributed with the source code. The input `sjeng.txt` is based on `458.sjeng/data/ref/input/ref.txt` with the depth (the even rows) changed from:

```
12, 13, 13, 12, 12, 16, 11, 13, 18
```

to:

```
6,  7,  7,  6,  6,  8,  5,  7,  8
```

The input `source.tar` for bzip2 is created with

```
$ gunzip -k bzip2-1.0.8.tar.gz
```

[8]The CVA6 repository is available at https://github.com/openhwgroup/cva6
[9]Details for the benchmarks are available at: https://github.com/dlgroupuoft/InFatPointer-ASPLOS2021-Eval.

```
# Olden
bh        4096 1
bisort    250000 1
em3d      2000 100 75 1
health    5 500 4
mst       1024 0
perimeter 10 0
power
treeadd   21 1 1
tsp       100000 1
voronoi   20000 1

# PtrDist
anagram   $FILES/words < $FILES/input.in
ft        10000 30000
ks        $FILES/KL-4.in
yacr2     $FILES/input2.in

# Others
benchmark -dh                       # wolfcrypt-dh
sjeng     $FILES/sjeng.txt
coremark
bzip2     -k -z $FILES/source.tar
```

Listing 3: Inputs to the Benchmarks

All the commands are run with `time -v` and the `stdout` redirected to `/dev/null`.

The functional evaluation of In-Fat Pointer follows a similar procedure above, although more manual configuration and analysis is involved. We use scripts to generate the Makefile, and all the binaries are run with a test script on the FPGA system. One can also test In-Fat Pointer with custom programs; a spatial error should result in a segmentation fault from dereferencing a poisoned pointer.

A.6 Evaluation and Expected Results

For the performance evaluation, Figure 10 and 12 should be reproducible from the spreadsheet generated by parsing the terminal log. For the functional evaluation, all applicable programs should pass the test.

A.7 Notes

The evaluation on Olden and PtrDist benchmark suites was added after the artifact evaluation procedure started, therefore their results are not independently verified for the reproducibility badges.

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 *Proceedings of the USENIX security symposium (USENIX Security '09)*. Montreal, Canada. https://www.usenix.org/legacy/event/sec09/tech/full_papers/sec09_memory.pdf

[2] ARM Ltd. 2020. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.* ARM Ltd. https://developer.arm.com/documentation/ddi0487/latest

[3] Todd Austin. 1995. Pointer-Intensive Benchmark Suite. http://pages.cs.wisc.edu/~austin/ptr-dist.html

[4] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994 Technical Conference*. USENIX Association, Boston, MA, USA. https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel

[5] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the ACM ASIA Conference on Computer & Communications Security 2018 (ASIACCS '18)*. Incheon, Republic of Korea, 381–392. https://doi.org/10.1145/3196494.3196540

[6] Martin C. Carlisle and Anne Rogers. 1995. Software Caching and Computation Migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*. Santa Barbara, CA, USA, 29–38. https://doi.org/10.1145/209936.209941

[7] J.Morris Chang, Woo Hyong Lee, and Witawas Srisa-an. 2001. A study of the allocation behavior of C++ programs. *Journal of Systems and Software* 57, 2 (June 2001), 107–118. https://doi.org/10.1016/S0164-1212(00)00122-9

[8] Dongwei Chen, Daliang Xu, Dong Tong, Kang Sun, Xuetao Guan, Chun Yang, and Xu Cheng. 2020. Saturation Memory Access: Mitigating Memory Spatial Errors without Terminating Programs. (April 2020). arXiv:2002.02831 [cs.CR] https://arxiv.org/abs/2002.02831

[9] Sourav Das, R. Harikrishnan Unnithan, Arjun Menon, Chester Rebeiro, and Kamakoti Veezhinathan. 2019. 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 (LCTES 2019)*. Phoenix, AZ, USA, 19–32. https://doi.org/10.1145/3316482.3326356

[10] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. Seattle, WA, USA, 103–114. https://doi.org/10.1145/1346281.1346295

[11] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Istanbul, Turkey, 487–502. https://doi.org/10.1145/2694344.2694383

[12] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. 2012. Baggy Bounds with Accurate Checking. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering Workshops (ISSRE '12)*. Dallas, TX, USA. https://doi.org/10.1109/ISSREW.2012.24

[13] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS 2017)*. San Diego, CA, USA. https://doi.org/10.14722/ndss.2017.23287

[14] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. Philadephia, PA, USA, 181–195. https://doi.org/10.1145/3192366.3192388

[15] Jason Evans. 2006. A scalable concurrent malloc(3) implementation for FreeBSD *(BSDCan 2006)*. Ottawa, Canada. https://papers.freebsd.org/2006/bsdcan/evans-jemalloc/

[16] Shay Gal-On and Markus Levy. 2012. Exploring CoreMark a benchmark maximizing simplicity and efficacy. (2012). https://www.eembc.org/coremark/

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

[18] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *Porceedings of the 37th IEEE Symposium on Security and Privacy (Oakland '16)*. San Jose, CA, USA. https://doi.org/10.1109/SP.2016.62

[19] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD 2017)*. Boston, MA, USA, 641–648. https://doi.org/10.1109/ICCD.2017.112

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

[21] Gnanambikai Krishnakumar, Kommuru Alekhya REDDY, and Chester Rebeiro. 2019. ALEXIA: A Processor with Lightweight Extensions for Memory Safety. *ACM Transactions on Embedded Computing Systems* 18, 6, Article 122 (November 2019), 27 pages. https://doi.org/10.1145/3362064

[22] Taddeus Kroes, Koen Koning, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2017. Fast and Generic Metadata Management with Mid-Fat Pointers. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec '17)*. Belgrade, Serbia. https://doi.org/10.1145/3065913.3065920

[23] Alexey Kukanov and Michael J Voss. 2007. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal* 11, 4 (November 2007), 309–322.

[24] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13)*. Berlin, Germany, 721–732. https://doi.org/10.1145/2508859.2516713

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code*

Generation and Optimization (CGO '04). San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[26] Hongyu Liu, Ruiqin Tian, Bin Ren, and Tongping Liu. 2020. Prober: Practically Defending Overflows with Page Protection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual, Australia) *(ASE '20)*. 1116–1128. https://doi.org/10.1145/3324884.3416533

[27] Google LLC. 2005. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[28] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. Orlando, FL, USA, 175–184. https://doi.org/10.1145/2544137.2544147

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Dublin, Ireland, 245–258. https://doi.org/10.1145/1542476.1542504

[30] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. 2019. FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19)*. San Juan, Puerto Rico, USA, 612–626. https://doi.org/10.1145/3359789.3359799

[31] NIST. 2017. Juliet Test Suite for C/C++. https://samate.nist.gov/SRD/testsuite.php

[32] 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, Article 28 (June 2018), 30 pages. https://doi.org/10.1145/3224423

[33] PULP Platform. 2019. pulp-platform/linux: Linux kernel source tree. https://github.com/pulp-platform/linux

[34] Marco Prandini and Marco Ramilli. 2012. Return-Oriented Programming. *IEEE Security & Privacy* 10, 6 (December 2012), 84–87. https://doi.org/10.1109/MSP.2012.152

[35] Hiroshi Sasaki, Miguel A. Arroyo, M. Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. 2019. Practical Byte-Granular Memory Blacklisting Using Califorms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Columbus, OH, USA, 558–571. https://doi.org/10.1145/3352460.3358299

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. Boston, MA, USA. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[37] Julian Seward. 2019. bzip2. (2019). https://www.sourceware.org/bzip2/

[38] Kanad Sinha and Simha Sethumadhavan. 2018. Practical Memory Safety with REST. In *Proceedings of the ACM/IEEE 45th International Symposium on Computer Architecture (ISCA '18)*. Los Angeles, CA, USA. https://doi.org/10.1109/ISCA.2018.00056

[39] 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 (Oakland '19)*. San Francisco, CA, USA, 1275–1295. https://doi.org/10.1109/SP.2019.00010

[40] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *Porceedings of the 34th IEEE Symposium on Security and Privacy (Oakland '13)*. San Francisco, CA, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[41] WolfSSL Inc. 2020. WolfCrypt Embedded Crypto Engine. (2020). https://www.wolfssl.com/products/wolfcrypt-2/

[42] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14)*. Minneapolis, MN, USA, 457–468. https://doi.org/10.1109/ISCA.2014.6853201

[43] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*. Beijing, China, 145–156. https://doi.org/10.1145/1755688.1755707

[44] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (November 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[45] Benjamin Zorn and Dirk Grunwald. 1992. Empirical Measurements of Six Allocation-Intensive C Programs. *ACM SIGPLAN Notices* 27, 12 (December 1992), 71–80. https://doi.org/10.1145/142181.142200