EFFICIENT SUBOBJECT-GRANULARITY SPATIAL MEMORY SAFETY ENFORCEMENT
WITH IN-FAT POINTER

by

Shengjie Xu

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto

© Copyright 2021 by Shengjie Xu
Lack of memory safety in programming languages like C and C++ can leave programs written in these languages with exploitable memory corruption vulnerabilities. Spatial memory safety defense can catch memory corruptions from out-of-bounds pointer arithmetic. However, none of the existing works can achieve low overhead, high compatibility, and fine-grained protection at the same time.

This thesis presents In-Fat Pointer, a hardware-assisted spatial memory safety defense that improves the protection granularity of existing tagged-pointer schemes using object metadata to subobject-bound granularity while maintaining their high compatibility and low overhead. In-Fat Pointer introduces multiple object metadata schemes to spare pointer tag bits from object metadata lookup, and use the spared bits with in-memory type metadata for subobject bound computation. The hardware prototype is implemented on an FGPA board, and In-Fat Pointer is evaluated in functionality, runtime and memory performance, and estimated hardware cost.
Acknowledgements

I would like to thank my supervisor, Professor David Lie, for his support and guidance throughout my research. I greatly appreciate his effort and advice on all aspects of my academic development. I also want to thank Wei Huang for his valuable support and feedback on this project, and I’m grateful to He (Shawn) Shuang, Lianying (Viau) Zhao, Rongzhen (Gavin) Cui, and Weicheng Cao for their feedback and discussion. I’m also grateful for the financial support from University of Toronto and Department of Electrical and Computer Engineering. Lastly, I would like to thank my family and friends for their support throughout my studies.
# Contents

1 Introduction ................................. 1  
  1.1 Contributions ........................................ 2  
  1.2 Thesis Structure ......................................... 2  

2 Background and Related Works ................. 4  
  2.1 Memory Errors and Attack Methods .................. 4  
  2.2 Tagged-pointers ................................. 5  
  2.3 Related work ........................................... 5  
    2.3.1 Fat-pointer scheme ................................. 7  
    2.3.2 Object-based scheme ............................... 8  
    2.3.3 Memory-based scheme .............................. 9  
    2.3.4 Tagged-pointer scheme for spatial memory safety .... 9  
  2.4 Implementation background ....................... 10  
    2.4.1 FPGA and HDL .................................. 10  
    2.4.2 RISC-V Instruction Set ......................... 10  
    2.4.3 Message Authentication Code ................... 11  

3 Design of In-Fat Pointer ....................... 12  
  3.1 Overview ............................................. 13  
    3.1.1 Promote Instruction and Metadata Organization .... 15  
    3.1.2 Poison bits ..................................... 16  
    3.1.3 In-Fat Pointer Registers ........................ 17  
  3.2 Object Metadata Schemes ......................... 19  
    3.2.1 Local Offset Scheme .............................. 20  
    3.2.2 Subheap Scheme ................................ 20  
    3.2.3 Global Table Scheme ................................ 21  
  3.3 Layout Table and Subobject Bound Narrowing .... 22  
    3.3.1 Layout Table .................................... 22  
    3.3.2 Subobject Index .................................. 26  
  3.4 Compiler Instrumentation ....................... 27  

4 Implementation Details .......................... 31  
  4.1 Metadata Format ..................................... 31  
    4.1.1 Global Table Scheme ............................ 32
List of Tables

2.1 Related Works Comparison ................................................. 6

3.1 Object Metadata Schemes Comparison ................................. 19
3.2 Layout Table Generation Rules .......................................... 27
3.3 Handling of Pointer Operations ........................................ 27

4.1 Layout Table Formats .......................................................... 34
4.2 List of New Instructions ...................................................... 39

5.1 Average Runtime of Evaluated Applications ......................... 66
5.2 Dynamic Instruction Count Composition of Evaluated Applications ........ 66
5.3 Average Memory Consumption of Evaluated Applications .......... 69
# List of Figures

2.1 Category of Related Work ........................................ 7
3.1 In-Fat Pointer Workflow ........................................... 12
3.2 In-Fat Pointer Bound Retrieval ................................ 13
3.3 In-Fat Pointer Instrumented Operations (High-level) ....... 14
3.4 In-Fat Pointer Instrumentation Overview (Code Example) ... 15
3.5 Metadata Organization and Promote Operation Overview .... 16
3.6 Pointer Form Conversions and Operations .................... 18
3.7 Local Offset Scheme .............................................. 20
3.8 Subheap Scheme .................................................. 21
3.9 Global Table scheme ............................................ 22
3.10 Layout Table Example ........................................... 23
3.11 Compiler Instrumentation for Pointer Operations .......... 28
4.1 Pointer Tag Formats ............................................... 32
4.2 Processor Micro-architecture Changes ......................... 43
4.3 Pointer Dataflow Graph Example ............................... 51
5.1 Promotion Performance Microbenchmark ....................... 61
5.2 Runtime Overhead on Evaluated Applications ................ 65
5.3 Dynamic Instruction Count Composition on Evaluated Applications .......................... 67
5.4 IFP Instruction Dynamic Count Composition on Evaluated Applications ................. 68
5.5 Memory Overhead on Evaluated Applications ................ 69
5.6 LUT Usage Increase in the Modified Processor ............... 71
5.7 LUT Usage Breakdown for IFP Unit ............................ 71
Chapter 1

Introduction

Memory corruption vulnerabilities in programs written in unsafe programming languages like C and C++ have been a research focus for more than a decade [41]. These languages do not strictly enforce the memory safety requirement that a pointer should point to the object from which it is derived from, therefore when such a pointer is dereferenced to access memory, errors can happen that cause the unintended memory to be disclosed or overwritten. This becomes a powerful primitive for attackers to mount further attacks, for example, code reuse attacks [34] and the more recent data-only attacks [19].

One important aspect is spatial memory safety which intends to stop the pointer dereference to access an unintended memory location because of bad pointer arithmetic. Bad pointer arithmetic can move a pointer outside its intended memory range, which would trigger a spatial memory error in the subsequent dereference. For example, an off-by-one error triggered by accessing the N-th element in an N-element array (where the valid index should fall into [0, N−1]) is a spatial memory error, and a scheme enforcing spatial memory safety should detect this error.

While many solutions have been proposed to improve the spatial memory safety for C and C++ code, they often face challenges or shortcomings in overhead, compatibility, or protection granularity. For example, many software-based solutions may impose a performance overhead as high as 115% [13], which has motivated solutions that rely on hardware acceleration [32, 44, 11]. Compatibility is also a challenge for many solutions. To check the safety of a memory access, the value of the pointer must be checked against pointer metadata, which describes the memory range over which the pointer may safely point [41]. However, these solutions either store that metadata along with the pointer as an inline fat pointer, thus changing the size of the pointer and causing incompatibility with legacy code [44], or they store the metadata in a separate memory region, thus increasing the cost to access and maintain the metadata [29, 32]. Finally, many solutions do not provide fine-grained protection [30, 3, 24]. Ideally, the intended memory range associated with a pointer should exactly cover the subobject(s) the pointer can point to. For example, a pointer derived from an array embedded inside a struct should not point to any other member of the struct when the pointer is dereferenced. However, coarse-grained protection schemes may not restrict the pointer to the enclosing array, therefore they cannot detect such intra-object overflow. In this thesis, schemes that can detect intra-object overflow are said to have subobject-granularity spatial protection; these schemes can narrow the bound associated with a pointer when the program change it to point to a member of the original struct. A pointer to a subobject under the

\[^1\text{In this thesis, a subobject represent the smallest memory range scope that a pointer derived from it should not access any memory outside the scope. This includes C struct members, arrays, and their counterparts in C++}.\]
top-level object (e.g., a pointer to a member of a struct) is referred to as an \textit{interior pointer}.

Recent advances in \textit{tagged-pointer} schemes introduce a new approach for spatial memory safety enforcement \cite{24}. A tagged-pointer scheme utilizes the unused high-address bits on pointers to store a metadata tag on each pointer, and a memory safety enforcement mechanism using this approach can use the pointer tag to lookup a larger, per-object metadata stored in memory to retrieve the object bound for spatial safety enforcement \cite{30, 22, 6}. The use of shared per-object metadata eliminates the expensive pointer bound maintenance, and the pointer tag is shown as an effective approach for inexpensive per-pointer metadata without breaking compatibility. However, among all tagged-pointer schemes that maintain backward compatibility, no hardware-based solutions have been proposed, and software-only works either do not provide subobject-granularity protection \cite{30} or impose high overhead \cite{13}.

This thesis presents In-Fat Pointer, a hardware-assisted scheme for spatial memory safety enforcement that addresses the challenges of overhead, compatibility, and protection granularity through efficient usage of the pointer tag bits. Unlike prior published works that use a single scheme for object metadata placement and lookup, In-Fat Pointer uses three complementary schemes that enable a range of objects while minimizing the number of pointer bits required for object metadata lookup. This enables In-Fat Pointer to use the remaining tag bits to implement efficient narrowing of subobject bounds to prevent intra-object overflow. As far as this author knows, In-Fat Pointer is the first hardware-based scheme using a tagged-pointer approach that can protect against intra-object overflow while maintaining the same pointer size as legacy code. This thesis shows that In-Fat Pointer can provide spatial memory safety comparable with fat pointer schemes while preserving compatibility and with lower memory overhead.

1.1 Contributions

This thesis makes the following contributions:

- The design and implementation of In-Fat Pointer are presented, which is a hardware-assisted scheme that provides low overhead, legacy code-compatible, and fine-grained spatial memory safety enforcement.

- The three complementary metadata schemes that In-Fat Pointer use are presented, which can retrieve object-granularity bounds with efficient use of pointer tag bits.

- The layout table is presented, which is a mechanism that In-Fat Pointer uses to narrow object bounds to subobject granularity to provide fine-grained spatial protection.

- A prototype implementation of In-Fat Pointer is presented, in which a processor implementation is synthesized on an FPGA. The prototype is evaluated for its functionality, runtime and memory overhead, and the estimated hardware cost.

1.2 Thesis Structure

The remaining chapters of this thesis are organized as follows. Chapter 2 introduces the background information, including memory safety tagged pointers and related works. Chapter 3 elaborates on the design of In-Fat Pointer, including the metadata schemes and the layout table. Chapter 4 lists the implementation details, including metadata format, ISA extension, processor modification, and the
Chapter 1. Introduction

compiler instrumentation. The evaluation of In-Fat Pointer is then presented in Chapter 5 and followed by the discussion on limitations in Chapter 6. Finally, Chapter 7 concludes this thesis and proposes some future work of In-Fat Pointer.
Chapter 2

Background and Related Works

2.1 Memory Errors and Attack Methods

Memory errors happen when a pointer dereference reads or writes memory not belonging to the original object the pointer is derived from. Such unsafe dereferences can happen when bad pointer arithmetic moves the pointer out of the bounds of the object (spatial violation), or when the original object has been freed, leaving the pointer dangling (temporal violation). Defenses that stop both spatial violation and temporal violation are said to provide complete spatial memory safety protection. This work focuses mainly on spatial memory safety, aiming to thwart attacks that trigger the first scenario.

Memory errors are usually leveraged as an attack primitive for memory read/write to circumvent deployed protections, and the only way to completely protect programs is memory safety enforcement. The attacks with the highest possible impact are those that achieve some form of arbitrary code execution, where the attacker can take control of an application to execute arbitrary code from the attacker under the application’s privilege. Contemporary computer systems deploy DEP/W ⊕ X defenses to prevent attackers from injecting executable code into application data. On these systems, instruction fetch on writable memory pages will generate exceptions. To circumvent these defenses, Return-Oriented Programming (ROP) reuses existing code in an application to perform Turing-complete computation, given that there is an arbitrary memory write primitive that can corrupt a code pointer in the application memory. ROP works by (1) finding usable code snippets (gadgets) from the compiled binary that end in a return-like instruction, (2) faking a stack with return addresses that chain the gadgets together, and (3) start the attack by corrupting a code pointer (e.g., a return address) using the arbitrary write primitive. To defend against control-flow hijacking attacks like ROP, Control-Flow Integrity (CFI) defenses enforce that indirect call and return destination follow the constraints derived from the program’s static control-flow graph. However, it is shown that coarse-grained CFI can be bypassed, and the more recent Data-Oriented Programming (DOP) can achieve Turing-complete code execution on certain programs without corrupting control flow data. Other existing deployed defenses make exploitations harder but do not prevent them from happening. Therefore, the only way to eliminate these attacks is to enforce memory safety policies so that attackers have no memory read/write primitive in the victim program.
2.2 Tagged-pointers

On a 64-bit architecture, each pointer is 64-bit in size. However, the processor typically cannot use all 64 bits for addressing. For example, current x86-64 implementations only use the lower 48 bits of a pointer for addressing. The hardware requires that a 64-bit address must be in a canonical form, where the high 16 bits (bit 63 to 48) must be the same as the highest bit in lower 48 bits (which is bit 47). If this condition is not satisfied, the hardware will generate an exception when a memory access using such a malformed address is executed.

The term “tagged-pointer” used in prior works refers to any pointer that has a custom metadata tag at the unused high-address bits, and “tagged-pointer scheme” refers to any work that uses tagged-pointers. Canonical address checks are handled by either removing the tag before pointer dereference or modifying the hardware to ignore the tag. Note that “tagged-pointer” refers to the method of metadata storage, which means:

- The term does not constrain what purpose the metadata serves. For example, ARM Pointer Authentication [4] (PA) use tagged-pointers to enforce code pointer integrity. It attaches a crypto signature as Pointer Authentication Code (PAC) on the pointer, and the PAC is checked before dereferencing the pointer to ensure its integrity and authenticity. PARTS [20] use ARM PA for both code and data pointer integrity. Mid-Fat Pointer [22] use tagged-pointers for generic metadata handling.

- The term does not constrain what metadata is stored on the tag. The metadata may belong to the pointer, the pointed object, or a more coarse-grained subject (i.e., a subject with less semantic or characteristic information than objects).

Note that the integrity and authenticity of the pointer tags are not guaranteed on their own unless the scheme includes pointer integrity or complete memory and type safety enforcement. The pointer tags can either be corrupted by attackers, who only need to corrupt the entire pointer as normal application data, or by the application themselves. Legacy C/C++ applications may cast a pointer to an integer, perform arbitrary manipulation on the pointer, and then cast it back before dereferencing.

2.3 Related work

Because it is impossible to determine the safety of all pointer dereference at compile time in unsafe languages like C/C++, all memory safety defenses introduce additional metadata into the program to model the safety expectations and insert runtime checks before important events (e.g., pointer dereferences) to verify that these expectations are met. For spatial memory safety protection, the introduced metadata would store information about what memory range is safe for a pointer to access.

There is a wide design space for memory safety schemes that involve trade-offs between performance, compatibility, and protection granularity. Table 2.1 compares In-Fat Pointer with recent related works in methodology, protection granularity, compatibility, and memory feature requirements. The related works are grouped into (1) fat-pointer schemes, (2) object-based schemes, (3) memory-based schemes, and (4) tagged-pointer schemes. Note that certain schemes in the table support temporal memory safety, and comparison on temporal memory safety enforcement is out of the scope of this thesis.

1 All addressing in this thesis refers to virtual addressing.
Table 2.1: Comparison between In-Fat Pointer and related works on memory safety, including hardware-based defenses, software-based defenses, and sanitizers.

<table>
<thead>
<tr>
<th>Defense or Sanitizer</th>
<th>In-Memory Metadata Subject</th>
<th>Spatial Protection Granularity</th>
<th>Lost Compatibility (^2)</th>
<th>Required feature (^3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel MPX [32]</td>
<td>Pointer</td>
<td>Subobject</td>
<td>—</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>WatchdogLite [28]</td>
<td>Pointer</td>
<td>Subobject</td>
<td>—</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>SoftBound [29]</td>
<td>Pointer</td>
<td>Subobject</td>
<td>—</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>CHERI [44]</td>
<td>Pointer + Object</td>
<td>Subobject</td>
<td>$\hat{}$</td>
<td>—</td>
</tr>
<tr>
<td>Shakti-MS [10]</td>
<td>Pointer + Object</td>
<td>Subobject</td>
<td>$\hat{}$</td>
<td>—</td>
</tr>
<tr>
<td>ALEXIA [21]</td>
<td>Object/None</td>
<td>Object</td>
<td>—</td>
<td>$\hat{}/-$</td>
</tr>
<tr>
<td>BaggyBound(^4) [3] [12]</td>
<td>Object/None(^\dagger)</td>
<td>Object</td>
<td>—</td>
<td>$\hat{}/-$</td>
</tr>
<tr>
<td>PArichCheck [45]</td>
<td>Object</td>
<td>Object</td>
<td>—</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>AddressSanitizer [37]</td>
<td>Memory</td>
<td>Partial</td>
<td>—</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>REST [40]</td>
<td>Memory</td>
<td>Partial</td>
<td>—</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>Califorms [36]</td>
<td>Memory</td>
<td>Partial</td>
<td>$\hat{}$</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>Low-Fat Pointer [24]</td>
<td>None(^\dagger)</td>
<td>Object</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SMA [8]</td>
<td>None(^\dagger)</td>
<td>Object</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CUP [6]</td>
<td>Object(^\dagger)</td>
<td>Object</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>FRAMER [40]</td>
<td>Object(^\dagger)</td>
<td>Object</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>EffectiveSan [13]</td>
<td>Object(^\dagger)</td>
<td>Subobject</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ARM MTE [4]</td>
<td>Memory(^\dagger)</td>
<td>Partial</td>
<td>—</td>
<td>$\hat{}$</td>
</tr>
<tr>
<td>In-Fat Pointer</td>
<td>Object(^\dagger)</td>
<td>Subobject</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

\(^1\) The scheme use tagged-pointers.

\(^2\) Whether the work can detect spatial memory errors at the bound of subobject (Subobject), at the bound of object (Object), or the detection is conditional or probabilistic (Partial).

\(^3\) Whether the work requires a shadow memory (\(\hat{}\)) (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 (\(\hat{}\)).

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

The approach using tagged pointers with object metadata allows In-Fat Pointer to provide the same fine-grained protections as traditional fat pointers without depending on special metadata memory features, for example, hardware-based tagged memory [48] which requires changes to the memory system, or shadow space memory that requires additional memory management change. Tagged memory refers to an unaddressable memory (from a user-level program’s perspective) that has a small fixed ratio with the associated physical memory, and each memory location is direct-mapped to a tag memory location. Manipulation to the tagged memory usually involves new special instructions. A common choice for the tag size is a one-bit tag per 64-bit memory, and it is common to optimize tag memory access by modifying the cache design. Shadow memory refers to a distinct memory address range that each (usually pointer-sized) word in the application’s virtual address space has a mapping to the distinct memory range. Shadow memory can be implemented in software and may use a complex mapping (including levels of indirections) to map a virtual address to a shadow memory address. Hardware-based works using shadow memory usually do not change the cache design to optimize shadow memory access.
Chapter 2. Background and Related Works

Figure 2.1: General approaches used in related work for checking. A scheme can either obtain the bound of a pointer and then perform bound checking (fat-pointer or object-based scheme), or enforce derived memory access policies without the notion of pointer bounds (memory-based scheme).

Figure 2.1 summarizes the approaches that each category of schemes use to obtain the pointer bound when a check is needed. The following subsections will introduce each category of schemes in greater detail.

2.3.1 Fat-pointer scheme

Traditional fat-pointer schemes persistently associate a pointer bound to each pointer throughout the pointer’s lifetime and check the address against the bound before dereferencing. Therefore, the bound of each pointer can be adjusted individually and the bound can preferably only cover the intended subobject, as shown in Figure 2.1a. Maintaining the pointer bound introduces a trade-off between performance, memory overhead, and compatibility. HardBound [11] is a hardware-assisted defense that manages pointer bounds in a direct-mapped shadow memory. While the direct-mapped shadow memory has lower performance overhead, in the worst case, the shadow memory range needs to be twice the size of the application’s memory. Intel MPX [32] stores the metadata in a separate two-level directory structure to reduce the reserved virtual memory range. Unfortunately, the indirection of the directory structure introduces the performance bottleneck. CHERI [44] expands a pointer to a 256-bit capability which stores the pointer bound inline with the address. This inline fat-pointer approach has both low performance
overhead and relatively low memory overhead comparing with other fat pointers. However, the data layout change breaks the binary compatibility, and applications may have to modify any code that makes assumptions on the pointer size. Besides the pointer size, CHERI in addition introduces changes in the programming model that requires source code modification for the capability-based protection.

While it remains an unsolved problem for fat-pointer schemes to achieve low performance overhead, memory overhead, and high compatibility at the same time, the semantics-rich per-pointer metadata is proven useful for enforcing strong memory safety policies. Fat-pointer based spatial memory safety schemes can detect overflow at the subobject level because the bound of each pointer can be individually narrowed. The compiler can instrument bound narrowing operations when a pointer is used to derive another pointer to a subobject so that later overflow on the derived pointer cannot corrupt adjacent subobjects. Besides pointer bounds, fat-pointer schemes can also include additional metadata to enforce other security policies. WatchdogLite \[28\] and Shakti-MS \[10\] uses a lock-and-key approach to guarantee temporal memory safety, where each pointer also carries a value as “key” that is required to match with the “lock” value from per-object metadata when dereferencing. Therefore, if a pointer is dereferenced after the object is freed, the error can be detected because the key will not have a matching lock.

Note that because of the way In-Fat Pointer use fat pointers\[2\], this thesis either uses “traditional fat-pointer scheme” or “traditional fat pointer” to refer to works that persistently associate a pointer bound to each pointer, and uses “fat pointer” to refer to the local construct that combines a pointer and its associated bound and can be passed through function interfaces (i.e., call and return). The bound is not required to be persistently associated with the pointer. In addition, all examples and comparisons in this thesis assume that fat-pointer schemes always have the bound narrowing enabled.

### 2.3.2 Object-based scheme

As shown in Figure 2.1b, an object-based scheme uses a single object bound for all pointers pointing to the same object. The object bound is typically encoded in an object metadata stored in the memory, and the schemes either store the metadata in memory along with the objects they describe or use a 1-to-1 mapping from an object to its metadata \[3, 12\]. The object metadata is usually immutable over the lifetime of the corresponding object and can be shared among all pointers to the object. The object metadata is located completely by the object address, without the help from pointer metadata. BaggyBounds \[3\] stores the size of each object in a bound table in the shadow memory, and instrument bound checks at each pointer arithmetic. BaggyBounds pads each object to be power-of-two sized and aligned, and uses a fixed slot size as the minimum object size so that the index of the bound table entry of an object can be computed by right shifting the object address. While the original BaggyBounds enforces the power-of-two sized allocation bound, it was improved to enforce the exact object size in later work \[12\]. Instead of storing the object size, PAriCheck\[45\] store an object label in the shadow memory, and instrument each pointer arithmetic to verify that the result pointer maps to the same label before the arithmetic. These schemes tend to have lower overhead than fat-pointers but they cannot provide subobject-granularity protection.

One observation from the design of object-based schemes is that it is possible for user space protection mechanisms to encode information on pointers without using unused high address bits by manipulating the object placement at the allocation time. Software-based Low-Fat Pointer \[14\] divide the application’s virtual address space into regions and dedicate one region for each size class of objects, effectively encode

\[2\]This is described in Section 3.1.3.
object size in the used virtual address bits on pointers. In this thesis, such schemes are still categorized as tagged-pointers.

2.3.3 Memory-based scheme

Memory-based schemes enforce coarse-grained policies in terms of memory, which are derived from object allocations, types, or other higher-level programming language constructs [37, 40]. As shown in Figure 2.1c, the enforcement mechanism usually only guarantees that accesses to memory that are marked invalid are caught, regardless of the higher-level semantics. These schemes cannot enforce fine-grained safety policies, but they typically have lower performance overhead. AddressSanitizer [37] 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 quarantine zones that delays the reuse of freed memory location to assist in catching some temporal memory safety violations. Califoms [36] insert padding bytes that should never be accessed into C/C++ structs, and use tagged memory to distinguish these bytes from normal application data. The hardware will trap any access to these tagged bytes.

2.3.4 Tagged-pointer scheme for spatial memory safety

Tagged-pointers were used for spatial memory safety enforcement before the term “tagged-pointer” was even introduced. Hardbound [11] implementation stores metadata opportunistically on high-address bits of each pointer. Hardware-based Low-Fat Pointer [24] first use tagged-pointers systematically to encode the entire pointer bounds on pointer tags, at the cost of increased external fragmentation for object allocations. These early works directly encode the pointer bound into the tag, as shown in Figure 2.1d. However, because of the limited number of pointer tag bits, the pointer bound must undergo a lossy compression, therefore this approach can only store pointer bounds coarser than object bounds, and may introduce excessive padding between object allocations.

As shown in Figure 2.1e, recent tagged-pointer schemes for memory safety enforcement use the pointer tags to locate in-memory object metadata that is shared across all pointers to the object. CUP [6] uses a 32-bit tag on each 64-bit pointer to store a per-object capability ID, and the object metadata is stored in a table indexed by the capability ID. FRAMER [30] proposes a scheme that uses a 16-bit tag to locate a generic object metadata near the object, and it provides an example use of object-granularity spatial memory safety enforcement. EffectiveSan [13] re-purpose the software Low-Fat Pointer to facilitate object metadata lookup, and use the type information at pointer use sites to narrow the object bound to subobject granularity, as shown in Figure 2.1f. EffectiveSan uses the layout hash table as in-memory type metadata, and given the use type of the pointer and offset into the object, EffectiveSan can retrieve the subobject bound offset from the layout hash table. However, the lookup into layout hash table is inefficient for hardware implementation, and EffectiveSan will lose the subobject-granularity bound if the use type of the pointer is inaccurate (e.g., a void pointer). In summary, when comparing tagged-pointer schemes using object metadata with traditional fat-pointers, combining pointer tags with in-memory object metadata eliminates the per-pointer memory overhead at the cost of losing the ability for per-pointer bound narrowing. Comparing with tagged-pointers that directly encode pointer bound, these schemes can use larger object metadata to store a precise object bound, and the use of object metadata introduces some level of detection for temporal memory safety.
Chapter 2. Background and Related Works

There are also works on using tagged-pointers with memory-based metadata. ARM MTE [4] tags the memory of each object with a 4-bit tag, and store the same 4-bit tag on pointers to these objects. It ensures that the tag from the pointer and the tag in the memory match before the pointer dereference, as shown in the right half of Figure 2.1c.

2.4 Implementation background

The prototype implementation of In-Fat Pointer involves an instruction set extension based on RISC-V and a modified processor with the ISA extension support that can be synthesized on an FPGA. To assist in detecting corruption on metadata, the design of In-Fat Pointer embeds message authentication code (MAC) into metadata that is vulnerable to corruption. This section provides the background information on related concepts necessary to understand the prototype implementation.

2.4.1 FPGA and HDL

A Field-Programmable Gate Array (FPGA) is a device that can emulate a hardware design. It has a chip that pre-builds a lot of hardware building blocks including look-up tables (LUTs), flip-flops (FFs), block RAM, digital signal processing (DSP) units, I/O ports, and interconnects. The connection between all these resources and the configuration of each block on an FPGA chip (and therefore what is the hardware design being emulated) can be reconfigured after the FPGA is manufactured. The configuration data for an FPGA is called bitstream. Contemporary FPGA stores these configurations in RAM after they are loaded from a bitstream, and the configuration will be lost after power-cycling. An FPGA development board connects an FPGA package to memory (including non-volatile flash memory that can store a bitstream, and volatile memory like DRAMs), peripheral devices (e.g., JTAG, UART, and Ethernet), and other devices or interfaces on a circuit board to simplify the task of building a complete system.

To use an FPGA to test a hardware design, the design first needs to be written in a Hardware Description Language (HDL), for example, Verilog and VHDL, and then the design is converted to a bitstream using an Electronic Design Automation (EDA) tool. The relationship among the design source code, EDA tool, and the bitstream is similar to a software source code, compiler toolchain, and the compiled binary. Because the bitstream format is dependent on the exact FPGA device, developers usually use the EDA tool from the FPGA vendor.

2.4.2 RISC-V Instruction Set

RISC-V instruction set is an open-standard RISC ISA that the specification of RISC-V ISA is released under an open-source license. It is suitable as a base instruction set for research in computer architecture and hardware for the following reasons: (1) modifications on the RISC-V instruction set do not require explicit licensing because the ISA is open-sourced; (2) the RISC-V instruction set leaves available instruction opcode space for custom extensions[20], which is suitable for research that adds additional instructions; and (3) there are open-sourced HDL implementations of RISC-V capable processors that can be synthesized and run on an FPGA development board.

RISC-V instruction set uses a modular design, where the base ISA only defines a minimal integer instruction set. All other operations, including integer multiplication and division, atomic instructions, and floating-point operations, are defined as standard instruction set extensions. The base instruction
set and each of the extension is assigned a unique character, and combining all the characters gives all the
instructions that the platform supports. For example, if a processor supports RV64IMAC, then beside
the base 64-bits instruction set (I), it also supports integer multiplication and division (M), atomic
instructions (A), and compressed instructions (C). The base ISA uses a fixed 32-bit instruction size,
and the compressed instruction extension introduces 16-bit encoding that each 16-bit instruction has a
one-to-one mapping to a 32-bit instruction.

RISC-V uses a load-store architecture, and it has 32 registers for both general-purpose registers
(GPR) and floating-point registers (FPR). Each instruction that contains one source register and one
destination register can carry a 12-bit immediate operand. RISC-V only supports \texttt{Reg+Imm} addressing
modes for loads and stores. RISC-V standard calling convention uses 16-byte natural stack alignment.

RISC-V uses \textit{Control and Status Registers} (CSRs) to store or present all key information of the
execution states or the system in general. For example, it is not only used for execution privilege levels
and modes but also for hardware performance counters. Besides a human-readable name, each CSR is
also assigned a unique 12-bit CSR \texttt{address} (which can fit in the 12-bit immediate operand space), and
a single set of CSR manipulation instructions are used to access all the CSRs. Therefore, adding new
CSRs do not need to introduce new instructions to access them specifically.

2.4.3 Message Authentication Code

A Message Authentication Code (MAC) is a keyed cryptographic hash computed with (1) a secret key
that is unknown to attackers, and (2) a message that may be known to attackers. While the message
can have arbitrary length, the secret key and the computed MAC typically has a fixed width determined
by the algorithm. When the defender wants to protect the integrity\footnote{Data integrity requires that any corruption on the data should be detected before it is used.} and authenticity\footnote{Data authenticity requires that the data crafted by an unauthorized source should be detected and rejected.} of a message, it can create a secret key and compute the result MAC for the message. The MAC is stored and sent
along with the message. When the data is used, the recipient can read the message, recompute the MAC
using the secret key, and check if it matches with the received MAC. Proper implementation of MAC
will guarantee that the probability that an attacker can craft a correct MAC without knowing the secret
key is negligibly small.

In-Fat Pointer is designed to detect spatial memory errors and stop the program once a spatial error
is triggered. However, it may not detect other types of memory error (i.e., temporal errors) as well as
errors from uninstrumented binary code. Therefore, it is possible that the object metadata used by
In-Fat Pointer is corrupted by an undetected memory error, which renders the protection ineffective.
To defend against these corruptions, In-Fat Pointer embeds MAC into vulnerable object metadata to
assist in detecting corruption to these metadata. In addition, erasing the MAC upon object deallocation
also gives In-Fat Pointer some ability to detect temporal memory errors on objects with MAC-protected
metadata.
Chapter 3

Design of In-Fat Pointer

In-Fat Pointer (abbreviated as IFP) is a hardware-assisted spatial memory safety defense that aims to achieve both fine-grained protection and low overhead while preserving legacy code compatibility. It is designed to provide transparent protection for applications with source code available. Figure 3.1 shows the overall workflow of In-Fat Pointer. A modified compiler is responsible for instrumenting the protection code into applications, and the resulting code is then linked with a runtime library to support the instrumentation. The output binary can then run with protection on an environment with the OS and the processor modified to support In-Fat Pointer.

In-Fat Pointer use compiler instrumentation to check the memory access addresses before pointer dereferences to ensure it is in the pointer bound, similar to other fine-grained spatial memory safety defenses. The core novelty of In-Fat Pointer is on how the bound of each pointer is retrieved when a check is needed. Figure 3.2 shows the overall approach for the bound retrieval. In-Fat Pointer inherits the basic approach from tagged-pointer schemes using object metadata, where a pointer tag in high address bits of a pointer can assist in lookup of in-memory object metadata, which encodes the object bound. This approach is compatible with legacy binary because no memory layout is changed, and the object metadata lookup can be efficiently implemented in hardware. After the object bound is retrieved, unlike prior tagged-pointer schemes that either directly use the (coarse-grained) object bound for checking [30, 6], or use expensive subobject lookup mechanism with in-memory type metadata only [13], In-Fat Pointer use a subobject bound narrowing procedure with a combination of in-memory type metadata (named as layout table) and per-pointer metadata stored on the pointer tag (named as subobject index). This improves both the performance and the accuracy of subobject bound narrowing.
Figure 3.2: In-Fat Pointer’s approach to retrieve pointer bound from the pointer. In-Fat Pointer improves upon prior tagged-pointer scheme using object metadata by (1) using multiple object metadata schemes to save pointer tag bits for object metadata lookup, and (2) using the spared tag bits to assist in subobject bound narrowing.

To address the shortage of pointer tag bits due to the new subobject index field, In-Fat Pointer includes three complementary object metadata schemes where each scheme is designed for a category of objects and can therefore use fewer pointer tag bits to lookup the in-memory metadata. In this way, In-Fat Pointer improves the protection granularity of tagged-pointer scheme using object metadata to subobject bound granularity without compromising compatibility or performance.

In this chapter, the high-level design of In-Fat Pointer is presented. Section 3.1 gives a complete overview of the design and explains how each component in In-Fat Pointer works together. Then, each of the subsequent sections describes a major component of In-Fat Pointer. Finally, Section 3.4 elaborates on the compiler instrumentation.

3.1 Overview

In-Fat Pointer is designed for a 64-bit instruction set where the hardware does not use all available bits on a pointer for addressing. Instead of requiring the pointer to have canonical addresses where the top unused bits must be the sign extension of address, In-Fat Pointer uses them to store a pointer tag for custom metadata. The exact width of a pointer tag can be adjusted; the prototype implementation uses 16-bit metadata tags.

Figure 3.3 shows the high-level instrumentation by In-Fat Pointer, and Figure 3.4 provides a code example and shows how the software and hardware collaborate to protect the application code. The modified hardware introduces new instructions for accelerated pointer checks and tag updates. The modified compiler will (1) instrument allocation of possible pointed objects to create object metadata on allocation and attach pointer tags to pointers to instrumented objects, (2) replace pointer arithmetic code with new instructions that update the pointer tag in addition to the arithmetic on the address, and (3) identify all pointers that cannot be determined to be safe statically and instrument them for runtime checking. The object metadata will also be cleared when the corresponding object is deallocated. The

\footnote{The code example assumes that no inter-procedural information is utilized for instrumentation, in other words the instrumentation on main() treats foo() as a black-box and vice versa.}
Chapter 3. Design of In-Fat Pointer

Figure 3.3: In-Fat Pointer instrumented run time operations, represented as boxes with grey background. IFP_Register(), IFP_Deregister(), IFP_GEP(), and IFP_Check() represent high-level instrumentations made by the compiler. The pointer check can involve the pointer bound retrieval shown in Figure 3.2 if needed.

Example code walkthrough. The code from Figure 3.4 uses IFP-prefixed functions to represent instrumented code fragments. The object metadata initialization and cleanup code is represented by IFP_Register() and IFP_Deregister(). IFP_GEP() performs both the pointer arithmetic and the corresponding tag update. The pointer checking code is represented by IFP_Check(). In function main() from the code example, because the local variable boo can be accessed through pointers from foo(), the compiler will (1) instrument the allocation of boo so that it has the object metadata (not shown in the figure); and (2) create a new pointer to boo that has the correct pointer tag, which is shown as ptr initialized by the return value of IFP_Register() at line 8. The object metadata will be cleared by IFP_Deregister(ptr) at line 18 when boo goes out of scope. If the program calls malloc() or another dynamic memory allocation function to create an object instead of declaring a local variable, then the compiler will modify the call so that the runtime library can manage the allocation and its object metadata. Beside instrumenting boo, the compiler will find that the result of &boo.value is passed as a function argument to foo(), which means the pointer will be checked in foo() and it should contain the up-to-date tag. Therefore, the compiler replaces &boo.value with IFP_GEP() at line 12 and uses its result as the function argument to foo() to ensure that the passed pointer has the correct tag. The hardware provides specialized instructions to update the pointer tag along with the address computation. Finally, inside function foo(), the compiler cannot guarantee that the argument pointer is safe to dereference, therefore the pointer is checked with IFP_Check(), and the returned pointer is dereferenced instead. 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). Therefore, pointers returned from IFP_Check() is dereferenced directly, and either the pointer is found valid and the dereference is successful, or the dereference trigger an exception if the check fails and the pointer is poisoned. Poison bits are introduced with greater details in Section 3.1.2. When a pointer needs checking, a new promote instruction is used to recompute the
Chapter 3. Design of In-Fat Pointer

Figure 3.4: Code example of instrumentation and HW/SW collaboration by In-Fat Pointer. IFP_Register(), IFP_Deregister(), IFP_GEP(), and IFP_Check() are explained in Figure 3.3. offsetof() is a macro in C that computes the address offset from the base address of the parent type (struct Boo) to the member (value).

pointer bound from available metadata, and then the bound can be used for later checking. Section 3.1.1 below elaborates the procedure of promote and the metadata involved. More details on the compiler instrumentation is presented in Section 3.4.

3.1.1 Promote Instruction and Metadata Organization

In-Fat Pointer introduces a promote instruction that takes a pointer as the input operand, uses the dynamic state in the tagged pointer to access the in-memory metadata, and recomputes the appropriate pointer bound. It encapsulates the pointer bound retrieval operations shown in Figure 3.2 from the software. This design choice brings the following two benefits. Firstly, object metadata schemes that require long sequences of arithmetic instructions can be utilized efficiently because the decoding overhead is eliminated. Secondly, In-Fat Pointer can therefore efficiently employ multiple object metadata schemes and optional pointer bound narrowing during bound recomputation, which would translate to unreachable or dead code for software-only instrumentation. Section 3.2 describes the designs of multiple object metadata schemes. Section 3.3.1 describes the design of the layout table, the data structure used for optional pointer bound narrowing.

Figure 3.5 shows all the metadata components used in In-Fat Pointer and events with instrumentation, as well as the flow of information between the metadata components. When an object is allocated, In-Fat Pointer will capture the base address and size information and encode them in the object metadata in memory. The promote instruction will use this metadata to derive the object-granularity bound. In addition to object metadata, if the type information is available at the object allocation site and the type contains subobjects (e.g., the program allocates a C struct instance with multiple members),
then a layout table is created for the allocated type and a pointer to the layout table is included in the object metadata. The layout table will assist in narrowing the object-granularity bound to the currently pointed subobject during promote. After in-memory metadata is initialized, the pointer to the allocated object will be attached with a tag that the promote hardware can utilize to locate the object metadata. In-Fat Pointer ensures that the pointer tag is in-sync with the address part of the pointer by instrumenting all pointer arithmetic code to update the pointer tags. When the pointer is later used and require checking, the promote instruction (1) uses the pointer tag to retrieve the object metadata and recompute the object-granularity bound, and then (2) if the layout table is available, accesses the layout table to narrow the bound to the currently pointed subobject. When the object is deallocated, the object metadata will be cleared to assist in catching temporal errors. The layout table is generated at compile time, stored in constant memory pages to prevent corruption, and shared for all allocations of the same type. The computed bound and the corresponding normal-sized pointer is grouped in a construct called In-Fat Pointer Register (IFPR), which is elaborated in Section 3.1.3 below.

In-Fat Pointer stores multiple pieces of metadata on each pointer tag. The pointer tag is split between the following fields: (1) scheme selector bits to indicate which scheme is being used, (2) the poison bits representing the validity of the pointer, (3) scheme-dependent metadata that is used by the object metadata scheme, and (4) subobject index used with the layout table. The scheme selector bits are required because it is not known at the compile time which scheme a pointer will use. The poison bits mark invalid pointers to assist in checking, and is elaborated in Section 3.1.2 below. The scheme-dependent metadata assists lookup of in-memory metadata during a promote operation. The subobject index represents which subobject the pointer is pointing to, so that the hardware can narrow the bound to correct subobject for interior pointers using the layout table. Section 4.1 elaborates the tag bits assignment in the prototype implementation with greater details.

### 3.1.2 Poison bits

Poison bits are part of the pointer tags in the unused bits on a pointer. They encode whether the pointer they are attached to is in one of the three states: (1) valid, meaning that it points within the bound; (2) recoverable out-of-bound, meaning that it points outside the bound but promote can still find the correct object metadata and recompute the correct bound; and (3) invalid, which is any other state. Poison bits
are updated each time an operation changes the value of the underlying pointer, or when any check fails. For example, the `promote` instruction will set the poison bits if the object metadata is invalid, which may happen in a use-after-free (temporal memory error) scenario where the object metadata is already cleared by the deallocation code but a dangling pointer to the object is dereferenced. The support for the recoverable out-of-bound state is necessary because (1) C/C++ permits off-by-one pointers to be brought back in bound and dereferenced\(^2\) and (2) the compiler may create temporarily out-of-bound pointers in loop optimizations. The current implementation dedicates two bits for poison bits to ease implementation and debugging, while a future implementation may only use a single bit to distinguish the valid and recoverable out-of-bound scenario, and use a special bit pattern for the entire pointer tag to represent the invalid state of the pointer.

The poison bits serve two purposes. First, because two of the object metadata schemes (the local offset scheme described in Section 3.2.1 and the subheap scheme in Section 3.2.2) use the address portion of the pointer as part of their metadata lookup schemes, `promote` operations on pointers in these two schemes could access invalid metadata when the input pointer is out-of-bound and the computed metadata address is incorrect. By updating the poison bits for resulting pointers in each pointer arithmetic operations, In-Fat Pointer ensures that all out-of-bound pointers can be reliably identified and handled by `promote` hardware. Second, standard `load` and `store` instructions check the poison bits and will trap if the state of the poison bits is anything other than valid. This has the benefit of extending some amount of protection to legacy code, which would also use standard `load` and `store` instructions, and thus would trap if it dereferences an out-of-bound pointer that it received from instrumented code. Bound checks can also be hoisted without introducing false positives when the same pointer is dereferenced by multiple load and store instructions. Hoisting the bound check will therefore only cause the pointer to be poisoned earlier, and the exception is not generated unless the poisoned pointer is dereferenced.

### 3.1.3 In-Fat Pointer Registers

In-Fat Pointer introduces In-Fat Pointer Registers (IFPRs) as logical registers which (1) contains the entire normal-sized, 64-bit pointer; and (2) embeds all necessary metadata for later checks, which currently include the pointer bound\(^3\). The `promote` instruction therefore takes a 64-bit pointer as input and produces an IFPR value as output, as shown in Figure 3.5. In-Fat Pointer ensures that IFPR values can drop their bound metadata and recompute them using information available on tagged-pointers when these pointers are valid. Because of this property, IFPR values do not need to be stored in application-managed memory because storing the normal-sized pointer is sufficient. The name of this work is an abbreviation of Internal Fat Pointer because IFPR can be used for bound check in the same way as traditional fat pointers but they are internal to the code being instrumented; the compiler may only spill them to stack slots but not save them to any memory where a change in the memory layout due to the pointer size increase would break the binary compatibility. Because a normal-sized pointer is typically allocated in a general-purpose register (GPR), this thesis uses “GPR” or “GPR value” to refer to a normal-sized pointer value, as opposed to an “IFPR value”.

Figure 3.6 shows how pointer values are converted between GPR and IFPR form, and all the oper-

---

\(^2\)An off-by-one pointer points to the first byte after the object, and can be generated by taking the address of N-th element in an N-element array. The current implementation supports off-by-one pointers in all object metadata schemes, but other out-of-bound scenarios may not be recoverable in all schemes.

\(^3\)In-Fat Pointer may be extended in the future to add more checks beyond spatial memory safety, while the concept of IFPR would remain the same.
Figure 3.6: Pointer form conversions and other operations defined on pointers. Incoming pointers are promoted to IFPR and checked before dereferences. Comparing with normal-sized pointers, pointers in IFPR form carry the bound information and can perform bound narrowing and size checks in addition to pointer arithmetic.

The implementation only consider pointers that have non-trivial uses (e.g., dereferenced or modified by pointer arithmetic) for promotion. Pointer values with only trivial uses (e.g., being copied from one memory address to another) are not considered for promotion.

The demote operation is essentially a truncation, except that the poison bit will be updated if the pointer is too far away from the bound that later promote cannot recover the exact bound.

sizeof() is the standard C/C++ operator.

In the current design, one of the three object metadata schemes (the local offset scheme introduced in Section 3.2.1) need the scheme-dependent metadata to be updated for each pointer arithmetic.
Table 3.1: Object metadata schemes comparison. Cross marks mean the scheme imposes constraints on the metric.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>placement</th>
<th>size</th>
<th>scalable</th>
<th>Use scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Offset Scheme</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>Small Objects and Local Variables</td>
</tr>
<tr>
<td>Subheap Scheme</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>Heap-allocated Objects</td>
</tr>
<tr>
<td>Global Table Scheme</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>Fallback Scheme</td>
</tr>
</tbody>
</table>

1 Whether the scheme avoids imposing constraints on the object base address
2 Whether the scheme has no maximum object size limitation
3 Whether the scheme has no maximum objects count limitation

Bound narrowing. When a pointer to a subobject is derived from a pointer to a parent subobject, besides the pointer arithmetic, the bound of the derived pointer must be narrowed so that subsequent arithmetic can be checked with a more accurate bound. In-Fat Pointer therefore introduces new instructions for bound narrowing and subobject index updates for this purpose. The bound narrowing operation is a part of \texttt{IFP\_GEP()} in Figure 3.4.

3.2 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 (1) efficiently instrument the allocation and manage the metadata of objects by exploiting the property of target objects, and (2) be able to spare pointer tag bits for subobject information storage, therefore achieving finer-grained spatial memory safety protection than prior works.

Table 3.1 shows the comparison of three metadata schemes. The first column lists the name of the metadata schemes, and the last column lists the positioning of the scheme. Each of these schemes makes assumptions on the target objects and therefore imposes constraints on them. The “placement”, “size”, and “scalable” columns in the middle of the table show the constraints that each of the schemes imposes. The local offset scheme is designed for small objects, especially for local variables allocated on the stack. The subheap scheme is designed for heap-allocated objects that have varying sizes and allocation counts. The global table scheme is designed as a fallback scheme that the previous two schemes cannot handle, for example, large global arrays. While the local offset scheme cannot serve for large objects, the subheap scheme cannot support objects at fixed base address, these two schemes combined can handle most of the objects in the application, and the global table scheme can handle the remaining objects.

When a \texttt{promote} instruction is executed, after the scheme is selected by the tag, the metadata scheme needs to recover the object-granularity bound given the pointer and the in-memory metadata. The implementation can also store a pointer to the layout table for the allocated type to facilitate subsequent subobject bound narrowing.

Because the local offset scheme and the subheap scheme place the in-memory metadata near application data and there is a risk of confusing the application data with their metadata, In-Fat Pointer introduces a MAC facility to embed a pointer-sized MAC in the metadata so that their authenticity and integrity can be verified. The key for the MAC computation is stored in a new control register introduced by In-Fat Pointer and is initialized by the runtime library.
3.2.1 Local Offset Scheme

The local offset scheme is intended for small object allocations, especially stack-allocated objects. This scheme is designed to minimize the instrumentation code size for such allocations and to be general enough to handle all small objects, including global variables or heap-allocated objects.

Figure 3.7 provides an illustration of the local offset scheme. The object metadata is appended to each object, and both the object and metadata base addresses are aligned to a granule, which is an implementation-defined power-of-two size. The pointer tag contains an offset field (granule offset), which represents the offset from the current address to metadata in the unit of granules. Thus, the promote operation retrieves the metadata by adding the address with granule offset and retrieving the metadata. 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 the prototype implementation, the local offset scheme’s object metadata is 128 bits in size. Section 4.1.2 elaborates on the exact format and bits allocation of the metadata.

The advantage of the local offset scheme is that it has no constraints on 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 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. The prototype implementation uses a granule size of 16 bytes, and both granule offset and subobject index use 6 out of the 12 bits each from the pointer tag. 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. Most of the objects are expected to be smaller than the imposed size limit [47] [7].

3.2.2 Subheap Scheme

Subheap scheme is intended for heap-allocated objects and is tightly coupled with the dynamic memory allocator design. As shown from Figure 3.8, the subheap scheme places objects inside power-of-two-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. The memory inside the block is expected to contain an array of allocation slots where each slot stores a single object, and the
Figure 3.8: Subheap scheme. F is an implementation-defined function that maps assigned tag bits to memory block size and metadata offset. In the prototype, F is implemented as a lookup from a set of control registers.

The design of the subheap scheme is inspired by parallel memory allocator designs that group objects according to their sizes into size classes and perform allocation based on size classes [27, 15, 23]. The design ensures that these parallel memory allocators can be ported to make use of the subheap scheme, and the implementation should provide support for common slot sizes that can match the size classes used by these allocators.

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

3.2.3 Global Table Scheme

The global table scheme is designed to handle all objects that the previous two schemes fail to serve, at the cost of a limited capacity for the number of objects served. As shown in Figure 3.9, the global table scheme stores all object metadata inside a single metadata table (referred to as the global table), and the pointer tag stores an index into the table. The table’s base address is stored in a control register, and both the control register and the table itself are initialized and managed by the runtime library. 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 bound is requested, the hardware simply indexes into the table to retrieve the metadata. 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.

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 table index stored in the pointer tag. In the prototype implementation, all 12 bits are allocated for the table index, and thus no subobject index is on the pointer tag, and objects using this scheme cannot narrow pointer bounds to the subobject in \texttt{promote}. The size of each row in the global table is 16 bytes.

### 3.3 Layout Table and Subobject Bound Narrowing

For subobject-granularity spatial protection, spatial memory safety defenses need to get the most precise bound for a pointer requiring checks. When a pointer is promoted and a bound is needed, In-Fat Pointer uses the following two-stage approach to recompute the pointer bound on demand. First, the object-granularity bound is computed using the scheme-dependent method, and then \texttt{promote} hardware narrows it to the exact subobject bound by utilizing the following two metadata. First, the subobject index from the pointer tag represents a \textit{summary} of the sequence of adjustment of the current pointed (sub)object (and correspondingly its pointer bound). Second, In-Fat Pointer uses the \textit{layout table} to represent the subobject memory layout in the object and map the subobject index back to the sequence of pointer bound narrowing to refine the object-granularity bound during \texttt{promote}.

This section first elaborates on layout table in Section 3.3.1, then describes subobject index briefly in Section 3.3.2.

#### 3.3.1 Layout Table

Layout tables are created by the compiler whenever it can obtain or infer the type from an object allocation, and the type is a composite type with subobjects (for example a \texttt{struct} type in C/C++). Each layout table is a flat array of elements where each element represents a possible subobject that an interior pointer can point to, and the subobject index from the pointer tag is simply an index into the layout table. The text below uses C/C++’s type concept interchangeably with language-independent terms, for example, using a struct type from C/C++ to represent a composite type that can have a different name in other programming languages.

Figure 3.10 shows a complete example of layout table, where the code in (a) triggers the generation of layout table for type \texttt{struct S} shown in (c). This example is used throughout this subsection to
Chapter 3. Design of In-Fat Pointer

1 // type definition
2 struct S { // 0: struct S
3    int v1; // 1: S.v1
4    struct NestedTy {
5        int v3; // 3: S.array[].v3
6        int v4; // 4: S.array[].v4
7    } array[2]; // 2: S.array
8    int v5; // 5: S.v5
9 };
10
11 // object allocations
12 struct S* ptr = malloc(sizeof(struct S) * size);
13 struct S s;
14
15 struct S s;

(a) Code example. The object allocation code triggers the generation of the layout table for struct S.

(b) Subobjects memory layout and the ordering of events during bound narrowing. The boxes are concrete subobjects, and the numbers are subobject indices.

Figure 3.10: Example of Layout Table. Each element in the layout table store a tuple of <parent index, base, bound, element size>.
explain the design of the layout table. The example assumes that each int is 4-byte in size.

3.3.1.1 Representation

As shown in Figure 3.10c, the layout table embeds a tree structure where each member of the struct has an element in the layout table, and the element of members will store the index for parent type’s element in the parent index field. If a member is a struct, then all its members are recursively embedded in the layout table. For example, the element for S.array (#2) will store the index of S (#0) as the parent index, and the parent of S.array[].v3 (#3) is S.array (#2). In the text below, the recursive embedding of members is referred to as subobject nesting.

Meaning of layout table elements. To explain exactly what a layout table element represents from the program, the considerations on array support need to be explained first. Arrays in source programs introduce challenges for a layout table design to support them efficiently. While a possible layout table design can use each element to represent one concrete subobject that has distinct memory ranges (and pointer bounds as a result) as if all array elements are similarly expanded in the layout table recursively as struct members, such design is not suitable for practical use because it cannot efficiently handle the following situations:

- When a struct type contains an array of structs that has a large array size, the layout table has to duplicate the similar elements for each array element because each array element is a concrete subobject that has distinct bounds. This will lead to unacceptably large memory overhead of layout tables and quick depletion of subobject index bits from pointer tags.

- A program can use code like ptr++ (where ptr points to an array element) to iterate over an array in a performance-critical loop. If the subobject index has to be adjusted in each of the iterations, the design needs to either instrument additional instructions that can incur unacceptable slowdown, or introduce a new hardware instruction that performs both the arithmetic and the subobject index update, which over-complicates the instruction set design because the instruction will need to take an additional operand for the subobject index change.

Therefore, instead of using each layout table element to represent a concrete subobject, the current design uses each element to represent an abstract subobject, where there is no distinction between an array element and the entire array. For example, as shown in Figure 3.10b, there are two concrete instance of S.array[].v3 (which is S.array[0].v3 and S.array[1].v3 respectively) in each element of S.array, and they are both represented by element #3 (S.array[].v3). In this design, the layout table size does not grow with the array size because all array elements map to the same abstract subobject (and therefore the same layout table element), and iterating over S.array with pointers incur no overhead from subobject index adjustment because the subobject index is the same for all elements in the same array.

Fields in a layout table element. Besides the parent index field which is mentioned previously, each layout table element contains a base and a bound field, which respectively express the lower and
upper address as offsets from the parent element. For example, because \texttt{S.array[]\_v3} (#3) is the first member in each element of \texttt{S.array} (#2), the element \texttt{S.array[]\_v3} has the base of 0 and the bound is the base plus its size, which yields 4.

While the base and bound are sufficient to completely describe simple structs where all the members are non-arrays, they do not handle the case that one of the members of a struct is an array of another struct type. For example, computing the pointer bound to a struct member of an array element in \texttt{S.array} from Figure 3.10a cannot be properly handled in this naive design. When promoting a pointer derived from the expression \texttt{&s.array[1]\_v3}, because element #3 represents its base and bound as the \texttt{offset} from the parent \texttt{S.array[]} instance, the hardware will need to know how to compute the base address of its parent element (offset of \texttt{S.array[1]}) before computing the base and bound of \texttt{&s.array[1]\_v3}, which is not possible without additional metadata to describe the array element size of \texttt{S.array[]}.

To fill in this missing gap, each layout table element carries an \textit{element size} in addition to parent index, base, and bound. The element size is the array element size if the member is an array, and is the type size (\texttt{bound} − \texttt{base}) in other cases. This information assists the hardware to compute the base address of array elements so that the bound of its struct member can be computed.

### 3.3.1.2 Bound Narrowing

The layout table is designed so that a single subobject index from the pointer is sufficient to describe the necessary bound narrowing operations. The design ensures that (1) each layout table element corresponds to a bound narrowing operation; and (2) the nesting relationship of the elements corresponds to the order of bound narrowing operations. Because the nesting relationship is a tree structure and each node (layout table element) saves the index to its parent node, the path from the tree root (which corresponds to the top-level element) to any desired destination subobject can be completely described by the subobject index of the destination subobject. As shown in (b) and (c) in Figure 3.10, given the subobject index 3 that corresponds to \texttt{S.array[]\_v3}, the hardware can traverse through the tree edges along 3 → 2 → 0 to fetch all the layout table elements necessary for bound narrowing from the root to \texttt{S.array[]}.

To perform a subobject bound narrowing operation, given a “destination” subobject index from the pointer, the hardware first fetches layout table elements from the destination to the root, then applies the bound narrowing operations according to the elements from the root to the destination. Considering the bound narrowing for a pointer from \texttt{&s.array[1]\_v3}, which is the second concrete subobject with subobject index 3 from left to right in Figure 3.10b in this walkthrough, all addresses are relative to the base address of the object (\texttt{s}) and the text may use “offset” to represent the “address”. Before the subobject bound narrowing, the object bound is \texttt{[0, 24]} and \texttt{&s.array[1]\_v3} is at offset 12 into \texttt{s}. The element #0 corresponds to the object bound, and no bound narrowing is necessary for it in this case. After fetching the elements #3 and then #2, The hardware will:

1. Compute the pointer bound of \texttt{&s.array} (#2). Because the base and bound is 4 and 20, the bound of \texttt{&s.array} is \texttt{[4, 20]} into \texttt{s}.

2. Compute the base address of \texttt{&s.array[1]}. Given that the element size of #2 is 8 and the bound of \texttt{&s.array} is \texttt{[4, 20]}, the hardware does the following:

   (a) Subtracting the lower bound of parent (4) from the offset of current address (12) to get the offset into element #2, which is 8.
(b) Dividing the offset 8 with the element size 8 to compute the remainder, which is 0 in this case. This means that $\&s.array[1].v3$ has the offset 0 into $\&s.array[1]$, or in other words, $\&s.array[1]$’s base address is exactly $\&s.array[1].v3$.

(c) Subtracting the result offset 0 from the current address (the address of $\&s.array[1].v3$, which has offset 12) to get the base address of $\&s.array[1]$, which is also 12.

3. Compute the pointer bound of $\&s.array[1].v3$. Given that the base and bound of element #3 is 0 and 4 respectively, the bound of $\&s.array[1].v3$ is $[12 + 0, 12 + 4]$ which is $[12, 16)$, where the 12 is the base address of $\&s.array[1]$ computed from the last step.

All these operations are done in the sequential order because of the data dependency, and the operation sequence is longer if the depth of the tree structure in the layout table is larger.

Because the number of nesting levels directly impacts the performance of subobject nesting, the design of the layout table includes an optimization that reduces unnecessary nesting. If there is a struct with a member that is in struct type (i.e., it has subobjects) but is not an array of struct, then all its subobjects are expanded in the same hierarchy as the member. Using the example from Figure 3.10 if the type of S.array (#2) changes from struct NestedTy[2] to struct NestedTy, then element #3 and #4 will have their parent set to #0 and their base and bound expressed as an offset from struct S instead of S.array[], in other words the hierarchy of S.array will be flattened if it is no longer an array. After this optimization, the only possible intermediate nodes in the embedded tree are those representing an array of structs.

3.3.1.3 Generation

Table 3.2 lists the complete generation rules for the layout table. The layout table generation procedure is generally a depth-first traversal of the type hierarchy where:

- Each “array of struct” member creates a new intermediate node in the tree, with the struct member expanded recursively under this intermediate node;
- Each struct member creates a leaf node with all its members expanded recursively under the same parent of the struct member; and
- Each elementary type or array of elementary type is a leaf node.

3.3.2 Subobject Index

In-Fat Pointer keeps the subobject index in sync with the intended subobject by instrumenting all code locations that involve changes in the subobject index. Table 3.3 lists all the pointer operations and the semantics handling. In-Fat Pointer instruments the subobject address computations so that besides the pointer bound is adjusted as other fat-pointer-based defenses, the subobject index is also updated to point to the new object. The design of layout table guarantees that for every subobject (struct member) address computation, the increment of subobject index is the same for all possible layout tables that have the struct S embedded, no matter which top-level type embeds the struct S.

Because the subobject index is used to index into the layout table, the size of a layout table is limited by the bit width of the subobject index from the pointer tag. More bits assigned to the subobject index enables the use of larger layout tables for more complex types.
Table 3.2: Layout table generation rules. `sizeof` and `offsetof` are standard C operators and `typeof` return the type of argument.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address change</th>
<th>New bound size</th>
<th>Subobject index change</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ptr++;</code></td>
<td><code>addr += sizeof(S)</code></td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><code>&amp;ptr-&gt;memb</code></td>
<td><code>addr += offsetof(S,memb)</code></td>
<td><code>sizeof(memb)</code></td>
<td><code>idx += L(S,memb)</code></td>
</tr>
<tr>
<td><code>(struct T*)ptr</code></td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

1 “—” represents no change to the specified value
2 \( L(S,memb) = 1 + \sum_{k \in [0,k_m)} |L(typeof(E_k),0,0,0)| \) where `memb` is the \( k_m \)-th member inside struct \( S \), \( E_k \) is the \( k \)-th member inside struct \( S \), and \(|L|\) is the number of rows of a given layout table. \( L \) is defined in Table 3.2.

3.4 Compiler Instrumentation

In-Fat Pointer’s compiler instrumentation utilizes IFPR and new hardware instructions to facilitate fine-grained bound checking, and prepare the aforementioned metadata components to assist metadata fetching. It is responsible for maintaining both the per-pointer metadata on the pointer tag and the object metadata in memory.

Figure 3.11 shows the compiler instrumentation in response to pointer operations along the value flow of the pointer. The table below describes the instrumentation made by In-Fat Pointer and compares that with prior works including traditional fat pointers (e.g., Intel MPX[32]) and tagged-pointer scheme using object metadata only (e.g., FRAMER[30]), because (1) these two approaches represent the prior state-of-the-art schemes for spatial memory safety enforcement that maintains binary compatibility.

Table 3.3: In-Fat Pointer’s handling of pointer operations, given `struct S *ptr;`.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address change</th>
<th>New bound size</th>
<th>Subobject index change</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ptr++;</code></td>
<td><code>addr += sizeof(S)</code></td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><code>&amp;ptr-&gt;memb</code></td>
<td><code>addr += offsetof(S,memb)</code></td>
<td><code>sizeof(memb)</code></td>
<td><code>idx += L(S,memb)</code></td>
</tr>
<tr>
<td><code>(struct T*)ptr</code></td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

1 “—” represents no change to the specified value
2 \( L(S,memb) = 1 + \sum_{k \in [0,k_m)} |L(typeof(E_k),0,0,0)| \) where `memb` is the \( k_m \)-th member inside struct \( S \), \( E_k \) is the \( k \)-th member inside struct \( S \), and \(|L|\) is the number of rows of a given layout table. \( L \) is defined in Table 3.2.
Chapter 3. Design of In-Fat Pointer

Figure 3.11: Compiler Instrumentation in response to pointer operations. All symbols used in the figure are listed in the legend in the bottom of the figure and explained in the text.

with legacy code, and (2) the instrumentation of In-Fat Pointer is inspired from these two approaches. The code example uses the declared types from Figure 3.10 and the functions with IFP prefix that represent high-level operations are from Figure 3.4. The operations listed on the top of the figure include all operations that a spatial memory safety enforcement mechanism (including IFP and related works) should handle, including pointer creation, arithmetic (which may or may not require bound narrowing), and dereference. All the semantics involved in the pointer operations from the language implementation’s view are listed below the operations. The “+” sign on the metadata (e.g., bound and tag) highlights that they impose additional semantics on pointer operations on top of the pointer’s original address computation semantics. The dataflow of the pointer value, as shown in the figure from left to right, may come across multiple code locations, in other words, the operations on the pointer can be performed in different functions. The pointer value can also be stored to the memory and loaded back, for example through a direct copy in the source code or by functions like memcpy(). These obstacles for the compiler instrumentation to pass arbitrary metadata along the pointer value flow is represented by the code boundary in the figure. When a program is not protected by any spatial memory safety mechanism, the only semantics associated with the pointer operations are the address computation. This case is represented by the vanilla scenario in Figure 3.11. When a spatial memory safety scheme is applied to

---

9Pointer type casts are not shown because (1) In-Fat Pointer ignores them as noted in Section 3.3.2 and (2) as far as this author knows, prior works either explicitly state that they ignore pointer casts (e.g., HardBound[11]) or the handling of casts are not mentioned.
the program and the scheme introduces additional metadata that cannot fit in the pointer, the scheme
must define how the metadata is handled. Because all spatial memory safety schemes must perform
a check before pointer dereferences to detect memory corruption, all comparisons below among In-Fat
Pointer and prior works on instrumented operations are focusing on metadata creation and propagation;
the use of metadata (i.e., the bound checks) are similar in design and is therefore not compared.

**Instrumentation of two approaches from prior works.** For traditional fat-pointer schemes, as
introduced in the Background (Section 2), they persistently associate a bound for each pointer, and they
either use shadow memory or store the bounds inline with the pointer to propagate the bound along
with the pointer value flow. This is shown in the second row of Figure 3.11 and the bound passing
strategies are represented by the “guaranteed passing with special design” symbol in the figure. Besides
the bound passing, for each pointer arithmetic that requires bound narrowing, the instrumentation will
also instrument instructions to set a new bound for the pointer. Therefore, these fat-pointer scheme
instrumentation would (1) instrument bound setup or narrowing code upon pointer creation or certain
arithmetic; and (2) handle bound passing across code boundaries. For tagged-pointer schemes using
object metadata, because these schemes devote all pointer tag bits for object metadata lookup, the
tag does not contain per-pointer metadata and is therefore not modified throughout the lifetime of the
pointer value. Therefore, the instrumentation of these schemes only needs to (1) instrument near pointer
creation so that objects have in-memory metadata and the pointers have the tag, and (2) load the object
metadata when the pointer bound is needed for the checking before the dereference.

**Instrumentation of In-Fat Pointer.** In-Fat Pointer is unique in that the design of IFPR and sub-
object index inherits the bound narrowing operations from traditional fat-pointer schemes, and the use
of object metadata requires the setup and cleanup of these object metadata as in prior tagged-pointer
schemes using them. Besides, In-Fat Pointer need to instrument pointer arithmetic on pointers where
the tags must be up-to-date because the local offset scheme depends on the correct granule offset for a
successful promote. Therefore, In-Fat Pointer requires the compiler instrumentation to instrument all
scenarios listed below.\(^\text{10}\)

**Object allocation instrumentations.** These instrumentations are made at pointer sources and are
represented by `IFP_Register()` and `IFP_Deregister()` in the earlier examples from this chapter. Whenever
the address of an object is taken and the resulting pointer may later be used for a check, the compiler
instrumentation needs to (1) select one of the metadata scheme for the object and instrument the scheme-
dependent metadata initialization and cleanup so that later pointer checks can find these metadata, and
(2) set up the pointer tag for pointers whose downstream of value flow will need the correct tag for
checking. While object allocation instrumentations are also performed conceptually by other tagged-
pointer schemes using object metadata, the support for multiple metadata schemes and layout table is
unique in In-Fat Pointer. For optimizations\(^\text{11}\) and implementation considerations, instead of proactively
checking each object allocations for instrumentation, whether to instrument an object or not should be
determined by a use-based analysis that all object allocations necessary for instrumentation are identi-
ﬁed by tracing back from pointers whose values may ﬂow to pointer checking sites. For example, the

---

\(^\text{10}\) The listed three instrumentation scenarios are also used by schemes that incorporate both object metadata and pointer
metadata using fat pointers, for example, ALEXIA, although the instrumented operations will differ for different designs.

\(^\text{11}\) Ideally, the compiler should not instrument objects whose metadata will never be used for pointer checks.
allocation of the object \( s \) alone does not cause it to be instrumented; the compiler should determine to instrument it after it finds that the pointer \( \text{ptr1} \) will be checked later on, and \( s \) can be pointed to by \( \text{ptr1} \).

**Pointer update instrumentations.** These instrumentations are made whenever the pointer value is changed, and are represented by \texttt{IFP\_GEP()} in earlier examples. For all the pointers that may be used in later checks, the compiler must make sure their metadata are up-to-date before the check, or before the pointers reach a code boundary if it may be checked after crossing the boundary. Therefore, the compiler will need to replace the pointer arithmetic instructions for these pointers so that they get the tag updated. The current implementation replaces all these instructions in place. Comparing with the instrumentation for traditional fat-pointer schemes, In-Fat Pointer instruments not only bound narrowing operations but also other pointer arithmetic that do not narrow pointer bounds, because of the need to update the pointer tags. To be specific, the compiler will update the subobject index (and the bound if present) for each bound narrowing, and the granule offset for local offset scheme pointers for each address computation in the pointer arithmetic. The hardware provides new instructions to perform these operations. In the example from Figure 3.11, the third operation (deriving \( \text{ptr3} \) from \( \text{ptr2} \)) involves only the pointer arithmetic, while the the second operation (deriving \( \text{ptr2} \) from \( \text{ptr1} \)) involves a bound narrowing in addition to the pointer arithmetic. In-Fat Pointer will instrument both of these operations.

**Pointer check instrumentations.** These instrumentations are made at pointer use (dereference) sites and are represented by \texttt{IFP\_Check()} in earlier examples. The compiler must guarantee that for each pointer being dereferenced, either the pointer is already poisoned, or the bound for that bound is large enough for the access size of the dereference. For pointers that the guarantee cannot be proved statically, the compiler will instrument (1) a bound retrieval, either by \texttt{promote} or by passing the bound across the code boundary from the upstream of the value flow; and (2) a size check on the retrieved bound. The compiler should attempt to propagate the pointer bound (i.e., propagate the IFPR form of the pointer) across code boundary whenever possible if it reduces the performance overhead from the \texttt{promote} in later checks. This is shown as “opportunistic passing” in Figure 3.11. The current implementation tries to propagate the pointer bounds across function calls and returns unconditionally and drop the bound using the \texttt{demote} operation if the pointer is saved to memory. When considering possible optimizations, the compiler should also attempt to drop the pointer bound whenever the downstream of the value flow no longer requires checking on the pointer value, so that the potential cost of the pointer bounds from register spilling are reduced.
Chapter 4

Implementation Details

This chapter presents the prototype implementation of In-Fat Pointer on the 64-bit RISC-V instruction set. The In-Fat Pointer prototype implementation is based on an open-sourced RISC-V processor CVA6[46], and the compiler instrumentation is implemented on LLVM[25] 10.0. The CVA6 fork of Linux (based on kernel version 5.1.0) is modified to support IFP-protected user-level programs. The prototype assumes that 48 bits out of a 64-bit pointer hold address bits and the top 16 bits can be used as the pointer tag. The current prototype only supports protection for user-level programs, and only C programs are supported. However, note that both are only the limitation of the prototype implementation and the design of In-Fat Pointer can support protecting kernel-level code and C++ programs. Section 2.4 provides some background necessary to understand the implementation, notably the Control and Status Registers (CSRs) of RISC-V instruction set.

Section 4.1 elaborates on the pointer tag and in-memory metadata format. Then, Section 4.2 describes the implemented instruction set extension, with a detailed description of the IFPR organization and new instructions added. Section 4.3 covers the micro-architectural modification on CVA6. Section 4.4 is about the modifications made on the compiler, including a new LLVM IR pass and changes to the RISC-V backend. Section 4.5 briefly discusses the changes to the Linux kernel and the functionality of the runtime library.

4.1 Metadata Format

In this section, the metadata format of the pointer tag and all in-memory metadata for the current prototype implementation are elaborated. Note that the design of In-Fat Pointer has many parameters that can be tuned for specific applications or usage, and it is out of the scope of this work to find the optimal values for each parameter.

The remaining text in this section uses the bit indexing and slicing syntax from Verilog/SystemVerilog to represent bit positions and ranges of a value. All the bit positions are indexed from the least significant bit, and the least significant bit has the index zero. For example, assuming that there is a 64-bit field called value, then value[1:0] selects the lowest two bits from value, and value[63] is the most significant bit of value.

Figure 4.1 lists all the pointer tag fields in the current prototype implementation. Among the 16 pointer tag bits at top of a pointer, the top 4 bits are interpreted in the same way for all pointer values.
Figure 4.1: Pointer tag formats.

The highest two bits indicate the three poison states; bit 63 is the persistent poison bit to mark non-recoverable errors, and bit 62 is the temporary poison bit that marks a temporary out-of-bound condition of a pointer. The next two bits (bit [61:60]) are the scheme selector bits. They are zero if the pointer is a raw pointer, which means that it points to an object without metadata. A non-zero bit pattern selects one of the three metadata schemes used for the pointed object. The remaining 12 bits are for the object metadata scheme and the subobject index and are interpreted differently for each scheme.

The following subsections introduce the pointer tag fields decomposition and metadata format for each scheme in the order of increasing scheme selector values. This is also the order that the schemes get implemented.

4.1.1 Global Table Scheme

For a pointer using the global table scheme, all the next 12 bits after the scheme selector bits ([59:48]) are used for the table index. The metadata table is currently a page-aligned global array declared in the runtime library, and each array element is a pair of 64-bit addresses, the first address for the lower bound and the second for the upper bound. Because only the runtime library would initialize the control register storing the pointer to the table, and the table is not relocated throughout the lifetime of the process, there is no risk of corruption from temporal memory errors in the application code, therefore the table does not contain additional metadata for integrity or authenticity verification. An alternative implementation may extend the metadata with a MAC if corruption from uninstrumented legacy code is considered a valid concern.

The global table scheme introduced a CSR (named ifpstatus) for storing a pointer to the global metadata table. Because the table is page-aligned, the lowest 12 bits of the CSR are re-purposed for storing additional mode data. Currently, the lowest two bits are used in hardware and the rest bits are reserved:

- Bit 0 controls whether the IFP extension is enabled (1) or not (0). This bit will be set by the runtime library on application startup.
- Bit 1 is the no-promote bit. The implementation supports a no-promote mode where the promote instruction always produce an IFPR with infinite bound and finish in a single cycle, and this bit controls whether this behavior is needed. This mode is used solely for running evaluations. See Section 5.3 for more details.
4.1.2 Local Offset Scheme

For a pointer using the local offset scheme, among the 12 bits left, the top 6 bits ([59:54]) are used as the granule offset, and the lower 6 bits ([53:48]) are used for the subobject index, as shown in Figure 4.1. The implementation uses 16-byte in-memory metadata for each object, therefore the granule size is 16 bytes. Both the object and metadata will have a 16-byte alignment, which is the same as the natural stack alignment on RISC-V. The 16-byte metadata has the following fields:

- a 64-bit layout table metadata word in the lower 8 bytes
- a 64-bit object metadata in the upper 8 bytes:
  - The top 4 bits ([63:60]) are for software checking and not interpreted by hardware
  - The next 12 bits ([59:48]) are the object size
  - The lowest 48 bits are used for a MAC

The layout table metadata word wraps a pointer to the layout table with additional metadata. Section 4.1.4 elaborates on this with greater details.

For the object metadata, because the address of the metadata can be implicitly used for object bound computation, only an object size is necessary for computing the object bound. The rest of the bits in the object metadata are used to make sure that the metadata is not confused by application data. To ensure the authenticity and integrity of the metadata, a 48-bit MAC is included at the lower 48 bits of the metadata. The MAC is computed by a new instruction ifp_mac, and In-Fat Pointer introduces the ifp_key CSR for storing the key that is used to generate the MAC. The top 4 bits of the metadata are currently intended for software to mark different origins of MAC, so that one can find what scheme owns the MAC and for what purpose when debugging the In-Fat Pointer implementation.

4.1.3 Subheap Scheme

The current implementation of the subheap scheme introduces 16 64-bit CSRs, ifpblk0 to ifpblk15, each of which stores a possible configuration that describes the size and metadata offset of applied memory blocks. In each of these CSR, the metadata offset is stored in the lower 48 bits and the size is expressed in power of two and is stored in the upper 16 bits.

For a pointer using subheap scheme, among the 12 bits left on the pointer tag, the top 4 bits ([59:56]) are used as a configuration index to indicate which CSR store the block size and metadata offset, and the left 8 bits ([55:48]) are for the subobject index. When the memory allocator allocates an object inside a memory block, the allocator is responsible for setting the configuration index to point to the CSR applied to that block.

The shared metadata in each block is 32-byte in size and the layout is shown in Listing 4.1. All the metadata for object bound computation is grouped in the first 16 bytes, therefore the address computation can start immediately after the first 16 bytes are loaded from memory without waiting for the second 16 bytes. The lowest 8 byte describes the base and bound of the array of slots. The base is represented in 4 bytes as an offset from the block base address to the address of the first slot, and the bound is represented in 4 bytes as an offset from the base of the one-past-the-last slot to the base of the next block (with the same power-of-two size). Expressing the upper bound of the array of slots in this way saves the metadata size when the block size is larger than 4GB, in which case an offset into the
Listing 4.1: Subheap per-block metadata

```c
struct BlockMetadata {
    // offset from block base address to allocation array start
    uint32_t lb_offset;
    // offset from allocation array end to "next" block base address
    uint32_t ub_offset;
    // Reserved [63:55]
    // isTriple [54] and slotSize as power of 2 [53:48]
    // allocation size [47:0]
    uint64_t sizeWord;
    // layout table metadata
    uint64_t layoutTableMetadata;
    // Identification [63:60]: 4'b1000
    // reserved [59:52]
    // configuration index [51:48]
    // MAC [47:0]
    uint64_t mac;
};
```

block may not be expressible using a 32-bit value. The next 8 bytes store the size of an allocation slot in the upper bits and the size of each object in the lower 48 bits. The current implementation supports $2^n$ or $3 \times 2^n$ slot size, and slot size is expressed in terms of the $n$ (at bit [53:48]) and whether there is a factor 3 (bit 54). While the flexibility of slot size is insufficient to support all size classes used in contemporary parallel memory allocators [27, 15, 23] and will increase the memory fragmentation, this choice is a reasonable trade-off that limits the hardware complexity. The next 8 bytes are for the layout table metadata, and the last 8 byte is the MAC. To assist in detecting corruption on the metadata, the upper 16 bits spared from the 48-bit MAC are used for additional checking. When a pointer using subheap scheme is promoted, the hardware will verify that the bits [51:48] match with the configuration index on the pointer, and that the top 4 bits match against a magic value 0b1000.

### 4.1.4 Layout table

The layout tables are generated on-demand at compile time whenever an allocation with type information is available and the type contains subobjects. They are emitted by the compiler as global constant data in LLVM (which will be stored in read-only memory pages) to prevent corruption, and they are linked with LinkOnceODRLinkage so that multiple identical tables across different compilation units can be merged by the linker. Each table is a flat array of elements, and each element is a tuple of (parent index, base offset, upper bound offset, element size). While the implementation includes 4 layout table formats to opportunistically shrink the layout table size, in practice, only 2 of them are in use. Table 4.1 lists the two formats and the bit assignment. The compact form is the default for any type less than 128KB in size, and the compiler will fallback to the full form if the type is too large.

<table>
<thead>
<tr>
<th>Format</th>
<th>Size Per Row</th>
<th>Parent Index</th>
<th>Base offset</th>
<th>Upper Bound offset</th>
<th>Element size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact</td>
<td>8 Byte</td>
<td>(12b) [62:51]</td>
<td>(17b) [50:34]</td>
<td>(17b) [33:17]</td>
<td>(17b) [16:0]</td>
</tr>
<tr>
<td>Full</td>
<td>32 Byte</td>
<td>(12b) [59:48]</td>
<td>(48b) [47:0]</td>
<td>(48b) [111:64]</td>
<td>(48b) [175:128]</td>
</tr>
</tbody>
</table>
Object metadata schemes supporting layout table will store a 64-bit layout table metadata. This 64-bit word is organized as follows, from MSB to LSB:

- [63]: Is array allocation (1) or scalar allocation (0)
- [62]: Reserved
- [61:60]: Layout table format; Compact form: 00, Full form: 10
- [59:48]: Layout table max subobject index
- [47:3]: Address of layout table
- [2:0]: Reserved

All layout tables are at least 8-byte aligned, therefore the lowest 3 bits of layout table address is expected to be zero. The layout table metadata contains a maximum subobject index to catch out-of-bound layout table access. This may happen when the pointer undergo a bad cast, or the pointer tag is corrupted and the subobject index is invalid. Bit 63 of layout table metadata indicates whether the allocation is allocating an array. Assuming that the depth of the root element of the layout table is zero, if the allocation is an array allocation, then all the base and bound offsets for elements with depth = 1 represent the offset from the base address of the array element instead of the base address of the entire array, and the hardware will narrow the bound to the array element before accessing the current element indicated by the subobject index from the pointer tag. Considering the example in Figure 3.10 if the object allocation is `struct S array[2];` then bit 63 of the layout table metadata in `array`'s object metadata will be set, and when the hardware compute the subobject bound for a pointer to `array[1].v1` after getting the object bound of `array`, this bit will instruct the hardware to first compute the base address of `array[1]` and then applying the base and bound offset of layout table element `S.v1`, instead of directly applying the base and bound to the base address of the entire `array`, which would incorrectly yield the bound for `array[0].v1`.

### 4.2 Instruction Set Extension

In this section, the design of current ISA extension is presented. All new instructions introduced by In-Fat Pointer used the `custom-0` and `custom-1` major opcode which RISC-V reserves for custom extensions. They corresponds to 0b00010 and 0b01010 for bit [6:2] in a 32-bit instruction. All CSRs introduced by In-Fat Pointer has the CSR number ranging from 0x801 to 0x81f. In-Fat Pointer also introduces bound registers to form logical IFPRs.

Section 4.2.1 introduces the design of IFPR and calling convention extension. Section 4.2.2 lists all the new instructions introduced by In-Fat Pointer and elaborates their functionality and use cases. Section 4.2.3 lists all the CSRs introduced to RISC-V.

#### 4.2.1 IFPR and calling convention

RISC-V instruction set uses 32 registers for both general-purpose registers (GPR) and floating-point registers (FPR), and instruction formats are defined with 5 bits for register operands. In-Fat Pointer therefore also introduces 32 logical IFPRs to simplify the implementation.
For spatial memory safety defense, comparing with a GPR value, an IFPR needs to include a pointer bound in addition to the pointer value. There are three possible strategies for implementing IFPR:

- **Standalone**: implement IFPR physically as standalone fat pointer registers. Used by CHERI[^44].
- **Decoupled register pair**: implement IFPR logically as a register pair that couples a GPR with a bound register. The GPR and bound register association is made *explicit* by specifying the register number of both the GPR and the bound register on an instruction that takes IFPR operand. Used by Intel MPX[^32].
- **Shadowing**: implement IFPR logically as a register pair that couples a GPR with a bound register. The GPR and bound register association is made *implicit* by having one bound register for each GPR, and therefore the same register number is used to reference both the GPR and the bound. Used by HardBound[^11].

The decoupled register pair approach is not considered for RISC-V implementation because all instructions that read or write bounds need to specify the bound registers explicitly. Because RISC-V uses 32-bit fixed instruction size, almost all RISC-V instructions take at most two inputs while producing an output[^1] and no instruction encoding is designed yet for instructions that have more than one destination registers. Therefore, instructions that write both the pointer and the bound (e.g., *promote*) require a new instruction encoding, and any instructions reading or writing bounds will have to sacrifice instruction encoding space for the additional bound register operand. The instruction encoding space loss is considered undesirable. Therefore, only the standalone IFPR and the shadowing approach are considered.

In the first iteration of implementation, the standalone IFPR approach is explored, and the following observations are made:

- **Standalone** IFPR complicates binary-compatible ABI. When passing a pointer argument or returning a pointer at function call/return interface, to pass the bound with the best effort while maintaining binary compatibility with legacy code, the pointer needs to be duplicated in both the GPR (for legacy code) and the IFPR (for instrumented code), and the instrumented code need special instructions to merge their values to avoid picking up stale IFPR value from uninstrumented caller that only populates the GPR input.

- **Standalone** IFPR increases the hardware cost for simple processors supporting at most 64-bit datapath. Certain datapath width (e.g., store queue and internal forwarding from writeback to operand read) need to be 3X or 4X of the original width because of the increase in the maximum register size.

- When dereferencing a pointer in IFPR, the standalone IFPR implementation either needs new instructions to perform load or store directly, in which case the implementation would consume more instruction opcode space and complicates code generation[^2] or the IFPR needs to *demote* to GPR first before using normal load/store instructions, resulting in code bloats. The latter approach was used in the implementation and runtime overhead of around 30% was observed in a crafted micro-benchmark with a tight loop.

[^1]: Except *fmadd*, which takes three register inputs and produces one output
[^2]: LLVM RISC-V backend tries to fold sign extension and truncation operations into load and store; create another version of loads and stores would need to duplicate these optimizations as well
Because of these experiences, the shadowing approach is used in the latest implementation.

4.2.1.1 IFPR basic organization

In-Fat Pointer introduces 32 bound registers, each of which is a $2 \times 48$ bits value where the lower and upper address bound takes 48 bits each. Each GPR is implicitly paired to a bound register, and the compiler models the register pair as a logical IFPR. An empty or infinite bound refers to a bound with all zeros; raw pointers will get an infinite bound when being promoted, and In-Fat Pointer do not prevent them from accessing any memory. For each pointer passed across a function interface, the bound register associated with the GPR will contain the bound if it is available, or is cleared if not available.

To eliminate the need for explicit size checks in performance-critical loops, In-Fat Pointer defines two sets of IFPRs:

- Bound-checked IFPR, where size checks can be folded into load/store when used as address operand; and
- Normal IFPR, where explicit size checks are needed before dereferencing the pointer.

The semantics of load/store from the base RISC-V instruction set is also extended as follows. For a bound-checked IFPR, when its GPR is used as the base address operand of a load or store, if the associated bound is not empty, besides the check on poison bits, the load-store unit will also check whether the computed access range falls inside the valid address range. In other words, the load using a bound-checked address operand now include an implicit size check asserting that the access range falls in the bound. This implicit size check is not made if the GPR belongs to a normal IFPR. The criteria for selecting bound-checked and normal IFPR is described next.

4.2.1.2 Calling convention considerations

The challenge in extending the existing calling convention with backward-compatibility in mind is to be able to pass the correct pointer bound across function call or return without sacrificing performance. An instrumented function should not get a wrong bound for an uninstrumented callee’s return value, or an uninstrumented caller’s argument. There should also be callee-saved bound registers so that an instrumented function does not have to save all bound registers at each function invocation. The extension is designed with the following deduction:

- All GPR used to pass function arguments and return values should have their associated bound cleared by the hardware when the GPR is modified by existing instructions, so that an uninstrumented callee can return a pointer with no bound when the instrumented caller populates another bound in the same register used for return values.

- All callee-saved GPR should have the associated bound registers preserved by callee as well, and the hardware should not clear a bound register when the associated GPR is used in an uninstrumented function.

Based on an observation that none of the registers for function argument and return value passing is callee-saved, the In-Fat Pointer implementation selects bound-checked and normal IFPR using the following criteria:
Chapter 4. Implementation Details

- All IFPRs with a GPR that is not callee-saved in the original calling convention are bound-checked IFPR
- All IFPRs with a callee-saved GPR in original calling convention are normal IFPR

In addition, the following semantics are introduced in hardware:

- Bound-checked IFPR will have their bound cleared when an instruction writes to the GPR
- Normal IFPR will have the bound preserved when the GPR is modified

RISC-V only has one standard calling convention defined, so the choice of bound-checked or normal IFPR is hard-coded in hardware implementation. When In-Fat Pointer is implemented on an ISA with multiple possible calling conventions, the implementation may use a control register to encode the calling convention for hardware. For systems mixing multiple calling conventions in the same process, the bound-checked IFPRs would the union of bound-checked IFPRs for all used calling conventions.

4.2.2 New Instructions

All instructions introduced by current implementation use the custom-0 and custom-1 major opcode in the RISC-V opcode map. The 7 least significant bits of each instruction is therefore 0b0001011 and 0b0101011, respectively. All instructions under custom-0 use the R-type instruction format, where each instruction has two source register operands (rs1, rs2) and one destination register operand (rd). All instructions under custom-1 use either the S-type or I-type instruction format. Both formats have a 12-bit immediate operand (imm12), and S-type instructions have two source register operands but no destination register, while I-type has one source and one destination register.

Table 4.2 lists all the instructions added by In-Fat Pointer. Instructions in each major opcode are distinguished mainly by funct3 field (Instr[14:12]) that is common to all instruction formats, and instructions in R-type also have a funct7 field (Instr[31:25]) to distinguish instructions or variants. The implementation does not use the compressed instructions; all new instructions are 32 bits in size. The [R], [I], and [S] annotates the instruction format. The rest of this subsection describes each instruction and the usage in greater detail. Unless otherwise noted, R-type instructions have their funct7 set to zero. If an instruction takes a pointer input operand, then unless otherwise specified, it is the rs1 operand. When a single operation is available in both R-type instruction and I-type instruction (e.g., promote and promotei), the later text may use the name of the R-type instruction (promote in example) to refer to both instructions.

Promotion. The R-type promote and I-type promotei implements the promotion operation. Besides the pointer bound retrieval that takes a pointer as one operand, both instruction variants can also fold a size check immediately after the bound is retrieved, and the size is the other operand for promotion. The I-type variant is preferred if the size is a compile-time known constant that fits inside the 12-bit immediate.

\[\text{This behavior is only enabled when IFP extension is enabled and the processor runs in user mode. Currently, no bound clearing is made in kernel mode to prevent the kernel from mistakenly erasing the bounds on a context switch.}\]

\[\text{The S-type instruction format is designed for store instructions.}\]
Table 4.2: List of new instructions introduced by In-Fat Pointer. \textit{ifpmd}, \textit{ifpsub}, \textit{ifppoison} are differentiated with the \texttt{funct7} field from the R-type instruction encoding.

<table>
<thead>
<tr>
<th>\texttt{funct3}</th>
<th>[R] custom-0</th>
<th>custom-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>promote</td>
<td>[I] promotei</td>
</tr>
<tr>
<td>001</td>
<td>ifpdbg</td>
<td>[I] ifpdbx</td>
</tr>
<tr>
<td>010</td>
<td>ifpextract</td>
<td>[I] ldbnd</td>
</tr>
<tr>
<td>011</td>
<td>ifpmac</td>
<td>[S] stbnd</td>
</tr>
<tr>
<td>100</td>
<td>ifpmd, ifpsub, ifppoison</td>
<td>[I] ifpmdi</td>
</tr>
<tr>
<td>101</td>
<td>ifpbnd</td>
<td>[I] ifpbndi</td>
</tr>
<tr>
<td>110</td>
<td>ifpadd</td>
<td>[I] ifpaddi</td>
</tr>
<tr>
<td>111</td>
<td>ifpchk</td>
<td>[I] ifpchki</td>
</tr>
</tbody>
</table>

**Bound setup.** The R-type \textit{ifpbnd} and I-type \textit{ifpbndi} sets the bound of a pointer. The two operands are a GPR pointer and a bound size. The lower bound is set to the current address, while the upper bound is set to the address plus the bound size. When the bound size is zero, however, the bound will be cleared.

**Subobject index change.** The I-type \textit{ifpidx} is used to add a fixed constant (subobject index \textit{offset}) to the subobject index field for an IFPR pointer in local offset scheme or subheap scheme. It is a no-op for other pointers. It is also a no-op when the offset is zero. The addition uses saturated math; if the subobject index would overflow (likely due to bad pointer casts), it will saturate at the maximum possible value.

**Pointer arithmetic.** The R-type \textit{ifpadd} and I-type \textit{ifpaddi} adds an address offset to an IFPR, and change the granule offset for pointers in the local offset scheme. This arithmetic includes an implicit size check of 1 byte; if the current address is out of bound, the temporary poison bit is set, and if the address is brought back in bound, the temporary poison bit is cleared.

**Size check.** The R-type \textit{ifpchk} and I-type \textit{ifpchki} performs the size check. One operand is an IFPR and the other is the size. The compiler backend uses a zero-sized size check to implement a register move for IFPR.

**Bound load and store.** The I-type \textit{ldbnd} and S-type \textit{stbnd} implements the load and store of bound register’s value from/to the stack frame. Same to other load and store instructions, only \texttt{Reg+Imm} addressing mode is provided, and the \texttt{rs1} operand is the base address while the \texttt{imm12} is the offset. Because the applications are not expected to load or save IFPR from memory other than the stack frame, \textit{ldbnd} and \textit{stbnd} do not perform implicit size checks in order to reduce hardware overhead.

**Demote and field extraction.** The R-type \textit{ifpextract} extracts a value from the input IFPR. It has following variants distinguished by the \texttt{funct7} field:

- \textit{ifpextract.lb} (\texttt{funct7}=0): Extract the lower bound and sign-extend it
- \textit{ifpextract.ub} (\texttt{funct7}=1): Extract the upper bound and sign-extend it
• `ifpextract.demote (funct7=2)`: Perform the demote operation. If the pointer is not temporary out-of-bound, return the GPR pointer part as is. Otherwise, if promotion cannot compute the correct bound, then return the pointer with persistent poison bit set.

**Metadata tag manipulation.** While the RISC-V base instruction set contains all necessary instructions for shifting and bitwise operations, they are inefficient in terms of instruction count when manipulating the pointer tags. The R-type `ifpmd` and I-type `ifpmdi` is used to reduce number of instructions needed for pointer tag manipulation. The `rs1` input for both instructions is expected to be a pointer and the lower 48 bits are copied to `rd`. Note that unlike other instructions where the R-type and I-type variants only differ in input operand formats, `ifpmd` and `ifpmdi` has different semantics, each designed for a distinct set of scenarios.

`ifpmdi` is designed for compiler instrumentation where all metadata is known at compile time and can be encoded in the `imm12` operand by the compiler. All upper 16 bits of `rd` are either covered by the provided metadata or set to zero. It has the following two major modes:

  - `imm12[10]`: array (1) or scalar allocation (0)
  - `imm12[7:0]`: layout table max row count

- Object metadata scheme helper (`imm12[11] = 0`): this mode is used in both the object metadata and tagged pointer computation. The highest 2 bits of output (`rd[63:62]`) is copied from the “sign” bit of the address input (`rs1[47]`), and the next 2 bits (`rd[61:60]`) are specified by `imm12[10:9]`. These two bits in addition defines the purpose of this `ifpmdi` instruction. Currently only the following two variants are defined:
  - `imm12[10:9] = 10`: local offset scheme pointer preparation. This is used to form a tagged pointer to an object using local offset scheme. Because the subobject index is zero when pointing to the top-level object, this variant copies `imm12[5:0]` to the granule offset field (`rd[59:54]`) and set the object index (`rd[53:48]`) to zero.

`ifpmd (funct7[6:2] = 0)` is designed for accelerating object metadata scheme implementation for both the compiler instrumentation and runtime library, where not all metadata is known at compile time. No layout table helper mode is defined for `ifpmd` because all metadata of layout table is known at compile time and `ifpmdi` alone is sufficient. Similar to the object metadata scheme helper mode of `ifpmdi`, `rd[63:62]` is copied from `rs1[47]`, and `rd[61:60]` is copied from `funct7[1:0]` and `funct7[1:0]` also defines the metadata scheme it is serving for. The following `ifpmd` variants are defined: (6’d0 represents 6 bits of zero and 8’d0 represents 8 bits of zero)

---

5Note that in current implementation, only off-by-1 pointers will not set the persistent poison bit in this case.
Chapter 4. Implementation Details

<table>
<thead>
<tr>
<th>funct7[1:0], mnemonic</th>
<th>rs2 is ...</th>
<th>rd[59:48] is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: ifpmd.raw</td>
<td>(arbitrary metadata)</td>
<td>rs2[11:0]</td>
</tr>
<tr>
<td>01: ifpmd.global</td>
<td>Global table index</td>
<td>rs2[11:0]</td>
</tr>
<tr>
<td>10: ifpmd.local</td>
<td>Granule offset</td>
<td>{rs2[5:0], 6’d0}</td>
</tr>
<tr>
<td>11: ifpmd.subheap</td>
<td>Configuration index</td>
<td>{rs2[3:0], 8’d0}</td>
</tr>
</tbody>
</table>

**Pointer subtraction.** Ifpsub instruction (funct7 = 0x10) is used to implement the pointer subtraction operations in C/C++, in which the programmers to subtract two pointers to get an address offset between them. When the pointers carry metadata tags, the integer subtraction (\texttt{sub}) no longer produces an address offset; the metadata is also subtracted and the upper 16 bits of subtraction result will be wrong. Instead of subtracting the entire 64 bits as the original \texttt{sub}, ifpsub will subtract the lower 48 bits only and sign-extend it to 64 bits. Besides the pointer subtraction, ifpsub with rs2 equal to zero is used to clear the pointer tag.

**Explicit Poisoning.** Ifppoison instruction (funct7 = 0x20) is used to explicitly set the persistent poison bit of a pointer. This instruction is used when the application code produces a statically out-of-bound pointer.

**MAC computation.** The R-type ifpmac is used to compute the MAC for protecting the metadata of the local offset scheme and that of the subheap scheme. In-Fat Pointer use 48-bit MAC so that they can carry additional metadata and use the same instructions for pointer tag manipulation to manipulate the metadata with MAC. The lower 48 bits of rs1 contain a pointer to the address where the MAC will be stored, and this is used as a nonce to prevent the attacker from copying another valid MAC from a different address to overwrite the MAC undetected. The upper 16 bits of rs1 is then used for metadata on the upper 16 bits of the output. The rs2 operand is the message for the MAC computation.

**Debugging extension.** The R-type ifpdbg is introduced to debug the implementation. It is not an essential instruction and can be removed without affecting the functionality. The hardware implementation includes a buffer for storing hardware internal events and additional statistics counters, and this instruction is used to access these facilities. Software simulators can safely emulate this instruction by writing zero to the destination register.

### 4.2.3 New CSRs

This subsection lists all Control and Status Registers added by In-Fat Pointer implementation. All listed CSRs are 64 bits in size, and user-level code has read/write access, except new hardware performance counters that are read-only.

**ifpstatus** (address 0x801): This CSR stores (1) IFP enable/disable bit (ifpstatus[0]), (2) “no-promote” mode enable/disable bit (ifpstatus[1]), and (3) address of global metadata table for global table scheme (ifpstatus[48:12]). All remaining bits are reserved and should be zero.
**ifpkey** (address 0x802): This CSR stores the secret key for all MAC computation, including **ifpmac** instruction. Note that while this register should be set to a random number at the beginning of the process, in the current implementation it is left as zero to simplify debugging.

**ifpdbgs** (address 0x803): This CSR is used to control the tracing component for the **ifpdbg** instruction. It can disable, enable, and flush the trace by writing 0, 1, 2 respectively to this CSR.

**ifpbblk0 to ifpbblk15** (address 0x810 to 0x81f): Each of these CSRs store a configuration for the subheap scheme. The configuration index on the pointer tag selects one of these registers to load the metadata offset (lower 48 bits) and the memory block size (upper 16 bits stored as power of two).

**New hardware performance counters** The implementation introduces the following new performance counters for IFP specific events.

- **IFP_INSTR** (address 0xc11): total number of IFP instructions retired
- **IFP_ARITHMETIC** (address 0xc12): number of IFP ALU-only instructions retired. **ifpmac** is also counted.
- **IFP_LSU_INSTR** (address 0xc13): number of **lbdn** and **stbdn** retired.
- **IFP_PROMOTION** (address 0xc14): number of **promote** and **promotei** retired.

### 4.3 Hardware Modification

This section describes the processor modification for implementing In-Fat Pointer. The hardware prototype is built on an open-sourced processor CVA6\(^6\) (renamed from Ariane) that supports RV64IMAC instructions. It has a 6-stage in-order pipeline that can issue one instruction and commit two instructions per cycle. The source code is written in SystemVerilog. While CVA6 does not claim to fully support the F and D floating-point extensions, it does contain a floating-point unit that can execute floating-point instructions correctly.

The CVA6 repository contains a full System-on-Chip (SoC) that can be synthesized on an FPGA. The CVA6 processor is connected to a 64-bit AXI bus with a DDR3 controller for the 1GB DDR3 RAM, a UART controller for a console, an Ethernet controller, and an SD card controller in addition to other necessary components. The default bootrom load the first 16MB of the SD card content to the beginning of the RAM address range, and branch to it. The SD card is formatted such that the first 16MB is a buildroot image wrapped by a bootloader, which can start a full Linux system on the FPGA. Almost all hardware modifications for implementing In-Fat Pointer are inside the CVA6 processor module.

The micro-architecture changes made to the CVA6 processor are shown in Figure 4.2. The primary changes are as follows:

- A new register file for pointer bounds is introduced, and the pipeline is widened so that each instruction can optionally take a pointer bound operand and produce a pointer bound result. The scoreboard is also modified so that bounds can also be forwarded in the same way as GPR values.

\(^6\)Tracing control is not implemented in **ifpdbg** instruction because the instruction is implemented in the execution stage and cannot flush the processor pipeline, while flushing the pipeline from CSR write is easier to implement.
The current implementation introduces new modules colored with dark gray, and modified the modules colored with light gray.

- The integer ALU is extended to implement new instructions that only involve combinational logic.
- A new execution unit, IFP unit, is added to the execution stage. All the remaining instructions not implemented in the integer ALU are implemented in the IFP unit.
- The load-store unit is modified primarily to implement `ldbnd` and `stbnd`, and to support the IFP unit for memory read.
- The remaining modules in the processor, including instruction decoder, CSR unit, and performance counters, are extended to support new instructions or new features.

### 4.3.1 Load-store unit

The load-store unit is modified so that (1) the load unit can help the IFP unit to read data from memory; (2) when a load or store is executed, the pointer tag (but not poison bits) is excluded from canonical address check, and the implicit bound check is performed if needed; and (3) `ldbnd` and `stbnd` support are added.

**Supporting IFP unit for memory read.** The baseline CVA6 uses a small queue for load/store requests so that the load-store unit can signify the ready state early in a cycle and let the instruction
issue logic to continue to the next instruction, even if the load or store may encounter situations that require multiple cycles (e.g., TLB miss) at a very late time of the cycle. The queue is extended so that besides the instruction issue logic, the IFP unit can also enqueue memory load requests to the load unit. When the result is loaded, the data will also be forwarded to the IFP unit directly. Note that because an earlier promote instruction may queue memory loads after a later store, and getting the wrong metadata overwritten by the store, the instruction issue logic is modified so that no store will be issued when a promote instruction is in execution.

**Pointer checking in load/store.** When a load/store is executed, the address generation unit that computes the actual address from Reg+Imm operands is augmented so that the result address with the access size is checked against the input pointer bound. The canonical address check is also modified so that when the IFP extension is enabled, the pointer tag bits are not checked. Therefore, for pointers from the user-level code, a poisoned pointer will be considered as not conforming to the canonical address, and an exception will be generated.

**Bound load and store.** The baseline CVA6 has two read ports and one write port from the data cache to the load-store unit, and each of the port can handle 8 bytes of transfer per cycle. One port is devoted to the load unit, one port to the TLB walker, and the last one for the store unit. When multiple read ports are raising requests, one of the ports is granted access by an arbitrator. To implement ldbnd and stbnd without compromising throughput, one additional read port is added for the load unit so that it can load 16 bytes of memory per cycle by using two ports in parallel. At the same time, the cache controller is modified so that parallel accesses on the same cache set can be satisfied in the same cycle.

### 4.3.2 IFP Unit

The IFP unit contains all the component necessary to implement promote and promotei. Because promotion logic requires MAC computation, ifpmac is also implemented in the IFP unit to share the MAC computation module. The debugging facility tracks a lot of internal states of promotion logic, therefore ifpdbg is implemented in the IFP unit as well.

In the current implementation, the IFP unit can read 16-byte for each request to the load unit. However, the requests are not pipelined, and each request takes three cycles when there is no miss in TLB or the data cache.

#### 4.3.2.1 Object metadata scheme handling.

The IFP unit implements each of the three object metadata schemes in a distinct child module with an identical interface to the rest of the logic in the IFP unit. The layout table is also encapsulated in a child module. A main Finite State Machine (FSM) coordinates all resources in the unit, and when a promote operation is executed, it activates the scheme handling module, grant it access to other resources in the IFP unit, and waits for its completion. After the object metadata scheme is handled, if the layout table metadata is available, the main FSM then activates the layout table handling module to walk the layout table and narrow the bound.

---

7In the current implementation, no store will be issued when the IFP unit is busy, even when it is executing some other instructions not accessing memory (e.g., ifpmac).

8The addition of the data cache read port and the change to the cache controller is only an optional improvement and is therefore not listed as primary changes to the processor microarchitecture.
Subheap scheme implementation. Subheap scheme handling is more complicated than the other two schemes because of the more complex metadata. Assuming the pointer being promoted is not out of bound and the lower 48 bits is the current address, to compute the address bound for the object, the subheap scheme handler needs to:

- Find the block base address and the metadata address by (1) reading the block size from one of the ifpblk CSR selected by the configuration index on the pointer tag, (2) mask out the lower bits of current address to compute the block base address, and (3) add the metadata offset from the CSR to get the metadata address.
- Load the per-block metadata and verify its integrity.
- Start to compute the object bound after the first 16 bytes of metadata are loaded. The bound is computed by (1) compute the slot array base address and ensure the current address is not out of bound of the entire slot array, (2) divide the offset from the slot array base address to the current address with the slot size to determine the base address of the slot and the offset into the slot, and (3) the object lower bound is the slot base address and the object upper bound is the lower bound plus the allocation size.

The current implementation limits the slot size to be in forms of $2^n$ or $3 \times 2^n$. While the first form is easier to divide, the latter one requires a multi-cycle divider. The implementation includes a table-based high-radix divider that can compute 8 quotient bits per cycle.

4.3.2.2 Layout table walker

The layout table walker is activated after the object bound is computed when the subobject index is non-zero and the layout table metadata is available. The main FSM in the IFP unit will not start a layout table walk if the subobject index is zero (i.e., the pointer points to the entire object and no subobject bound is needed) or the layout table metadata is not available. To perform a subobject bound narrowing, the main FSM performs the following in order:

- If the top-level allocation is an array allocation (bit 63 is 1), the “object bound” from the object metadata scheme is the bound of the entire array instead of an array element. Therefore, to get the base address of the current array element, element 0 from the layout table is read first and the bound is narrowed to the array element. This step is skipped if the top-level allocation is not an array allocation.
- Starting from the element pointed by the subobject index from the pointer, fetch the layout table element, and if the element is not a direct child of element 0 (i.e., the parent index field is non-zero), the parent element is fetched next. All the fetched elements except the last one are pushed to a register-based stack.
- Perform the bound narrowing in the reverse order the elements are fetched.

The register-based stack for saving layout table elements temporarily is necessary because the bound narrowing operations must be replayed from the root element to the leaf element, which is in reverse order of fetching.
The bound narrowing operation is handled in a child module inside the layout table walker. The
bound narrower module maintains a notion of the “current most-narrowed bound” expressed as offsets
from the object lower bound, and each replayed bound narrowing operation will adjust the offsets. There
are two possible narrowing operations:

- array element bound narrowing: knowing that the current bound is the bound of an array, compute
  the bound of the current pointed array element given the element size.

- struct member bound narrowing: knowing that the current bound is the bound of a struct in
  C/C++ notion, compute the bound of a member given the base and bound offset from the struct
  base address.

Whenever a layout table element is fetched from the memory, it is “decoded” into one or both of the
bound narrowing operations above. Then, after all the layout table elements are fetched, the FSM in
the layout table walker pops the operations from the stack and activates the bound narrower to perform
the operations. Note that while the struct member bound narrowing is single-cycle, the array element
bound narrowing requires the use of a divider that is likely multi-cycle. The current implementation
uses a simple shift divider that produces one bit of quotient per cycle.

The layout table walker is the most complex module in processor modification. When comparing the
number of lines of (manually written) code without comment or blank lines, the layout table walker has
1,030 lines of SystemVerilog while the three object metadata schemes have 676 lines of code in total. In
comparison, the original integer ALU has 104 lines of code and the modified one has 284 lines of code.

4.3.2.3 MAC Computation.

The IFP unit contains a MAC computation module that produces a 48-bit output given three 64-bit
inputs: a key, a message, and a nonce (where the lower 48 bits are the address where the MAC is stored
and the upper 16 bits are uninterpreted metadata). The IFP unit implementation makes no assumption
on the number of cycles the MAC would take. The main FSM in the IFP unit will coordinate the input
to the MAC computation module so that both ifpmac and promote logic can access them when needed.
For prototyping purposes, currently the MAC is a simple XOR of inputs with bit position shuffled. For
production uses, the MAC can be implemented using light-weight ciphers like QARMA[5].

4.3.3 Debug facility

The implementation of In-Fat Pointer started before gdb or gdbserver can work natively on RISC-
V architecture, and the software toolchain was stuck in old versions so that little debugging help is
available. The CVA6 implements the debugging extension and one can use OpenOCD to debug the
software through the JTAG interface, but the debugging utility is insufficient and breakpoints do not
always work. Therefore, besides the functional modification, two optional debugging modules are also
implemented to assist in debugging the implementation:

- A tracing module inside IFP unit implements ifpdbg instruction. It logs the execution history of
  IFP introduced instructions to assist in debugging the software.

- A halt detection module is implemented outside the CVA6 processor to debug the hardware imple-
  mentation. It is interposed between the UART controller and the AXI bus. In normal conditions,
it transparently passes the communication between the AXI bus and the UART controller, and it keeps a trace of selected internal signals from the CVA6 processor. When the module finds that the processor stopped committing instructions for a configured number of cycles, it will take over the UART controller and dump the recorded trace near the last time the processor committed an instruction.

4.4 Compiler Modification

The Clang/LLVM 10.0 is modified to instrument the application programs and generate new instructions. The changes include the following:

- New intrinsic functions are added to LLVM IR to represent IFP operations, and some of them are declared as Clang builtin functions so that the runtime library can use them to generate specific instructions when compiled by the modified Clang toolchain.

- An LLVM IR analysis and instrumentation pass is added after all the other IR optimizations complete. This IR pass implements all the analysis and instrumentations, as well as some implementation-independent optimizations.

- The RISC-V backend is modified so that it (1) can generate new instructions and make use of new registers; (2) implements target-specific optimizations for the instrumented instructions.

- Some LLVM IR optimization passes (\texttt{InstCombine} and \texttt{IndVarSimplify}) are modified to preserve pointer and type information when IFP is enabled. This is necessary to fix false positives observed on certain test programs.

Section 4.4.1 elaborates on the implementation of the LLVM IR pass. Section 4.4.2 describes the modification on the RISC-V backend.

In the text below, the \textit{analysis domain} refers to all the code that the LLVM IR pass can analyze and instrument in the single LLVM module the IR pass runs on. This domain excludes (1) functions that are explicitly annotated to ignore, and (2) functions that the current IR pass implementation does not support (e.g., variadic functions). A pointer is considered \textit{escaped} if its value can be obtained from outside the analysis domain through a memory location, or a call or return. Note that other works may use the term \textit{escape} to describe value flow crossing a function boundary, while in this work it describes the pointer value flow crossing the analysis domain boundary.

4.4.1 IR Analysis and Instrumentation Pass

As described in Section 3.4, the LLVM IR pass for In-Fat Pointer (short-named as IFP pass below) needs to (1) instrument object allocations and deallocations to setup object metadata and pointer tag, (2) instrument pointer updates to keep pointer tag and bound up-to-date for checking, and (3) instrument pointer checks with \texttt{promote} and size checks. The entire procedure is performed in the following stages:

\texttt{IndVarSimplify} can optimize away redundant induction variables of each loop by preserving one of the variable and express the rest using the chosen variable. In case there are multiple pointers with identical value but different type as induction variables, \texttt{IndVarSimplify} is modified to favor the pointer with the largest type instead of blindly choose the “first” one according to the (somewhat arbitrary) order of LLVM IR instructions as the induction variable.
---

**Chapter 4. Implementation Details**

- Preprocessing: perform special handling for certain code patterns in input IR to prevent miscompiling troubling code and simplify the implementation in the rest of the pass.

- Graph-based Analysis and Instrumentation: the pass builds an inter-procedural pointer dataflow graph that describes the value flow of data pointers, and then perform a (control) flow-insensitive static analysis on this graph to generate instrument decisions. Finally, the instrumentation phase goes over all the graph nodes to perform the instrumentation decision.

The Clang frontend is modified to run the IFP pass after all optimization passes when the IFP feature is enabled, and it also runs additional optimizations\(^\text{10}\) after the IFP pass to further clean-up the instrumented code. Note that the current implementation use pointers pointing to address space \(^{254}\) to represent IFPR pointers; normal-sized pointers points to address space 0. Most of the new instructions are introduced in LLVM IR as intrinsic functions as recommended by the LLVM IR extension guideline \(^{35}\).

This subsection is organized as follows. Section 4.4.1.1 describes the special handling performed on the input IR. Section 4.4.1.2 to 4.4.1.5 elaborates on the graph-based analysis and instrumentation.

The In-Fat Pointer IR pass is one of the most complex components in the In-Fat Pointer implementation. At the time of writing, the entire IR pass contains about 10k lines of code excluding comments and blank lines.

**Facts and assumptions relied on by the instrumentation implementation.** When a pointer requires checking, the pointer should be in IFPR form with the bound available for checking. The compiler can create IFPR for checking with \texttt{ifpbnd} instructions if the pointed object can be statically identified. Otherwise, to get the IFPR, the compiler either depends on \texttt{promote}, or expects IFPRs being passed through function calls and returns. As mentioned in Section 3.4, the current implementation always tries to pass the IFPR across function calls and returns whenever possible. The IR pass assumes that \texttt{promote} instructions are expensive and are implemented to try to reduce number of \texttt{promote} instructions.

### 4.4.1.1 Preprocessing

After the optimized IR input is received, the first task of the IFP pass is a preprocessing stage that performs the following operations.

**GetElementPtrInst simplification.** The LLVM IR uses a single \texttt{GetElementPtrInst} (or \texttt{gep} in shorthand) to represent all pointer arithmetics including struct member address computation and array element address computation, and a single \texttt{gep} can contain more than one such member address computation operations and they can mix in arbitrary orders. However, IFP needs to instrument bound narrowing and subobject index change for struct member address computation, and therefore needs to distinguish the “elementary” address computations from the mixed \texttt{gep}. The preprocessing stage guarantees that all the \texttt{gep} after preprocessing is either (1) one or more “unsafe” arithmetics, which corresponds to array element address computation, or (2) one or more “safe” arithmetics which corresponds to struct member address computation. All \texttt{gep} mixing these two types of operations are broken down into multiple \texttt{geps}.

---

\(^{10}\) In the current implementation, this includes LICM, GVN, and DCE.

\(^{254}\) LLVM include a backend-specific address space (an integer in the range of \([0, 255]\)) as part of the pointer type.

\(^{35}\) Section 5.2 and 5.3.1.1 evaluates the performance overhead imposed by \texttt{promote} instructions.
**Pointer subtraction handling.** The LLVM IR does not have pointer subtraction instructions. The C/C++ code that performs pointer subtraction is lowered into an IR that first cast pointers to integers, then subtract the values. This can cause wrong subtraction results after instrumentation: if any one of the pointers contains a pointer tag, the integer subtraction result will include the difference of pointer tags in addition to just the address offset, and is likely to lead to program misbehaving. To address this problem, the preprocessing stage identifies all pointer subtraction patterns in the input IR, and replaces them with calls to a new intrinsic function `riscv_ifp_diff_gpr` which would lower to `ifpsub` instructions in the backend.

**Pointer value manipulation handling.** Sometimes the application may perform manipulations on the address part of the pointer for aligning data structure, for example allocating a 32-byte buffer and mask off the lowest 4 bits to get a 16-byte aligned buffer. When represented in the IR, the code will first convert the pointer to an integer (using `ptrtoint` instruction in LLVM IR), then apply the operations, and finally cast the result back (using `inttoptr` instruction). This case also requires special handling because of the following two reasons. Firstly, the arithmetics can cause the pointer tag to be stale. For example, the granule offset of a local offset scheme pointer may have a wrong value if some arithmetic is not performed by the instrumented pointer arithmetic instructions. Secondly, casting the pointer to integer and cast manipulation result back to a pointer can cause the later analysis to believe that the pointer before manipulation will escape, therefore if the pointer is from a local variable declaration, the cast will force the local variable to be instrumented even if it will not have any pointers escaping. To fix these issues, the preprocessing stage first identifies all such code patterns, and then replaces the last `inttoptr` with an instruction sequence that (1) computes the offset introduced by the pointer manipulation from the result of `ptrtoint` to the input of `inttoptr`, and (2) replace the `inttoptr` with a `gep` that add the computed offset to the address. In other words, while the old code directly manipulates the pointer value which looks like a black box to the later analysis code, the new code expresses the address change as pointer arithmetic explicitly. In this way, the pointer tags of the source pointer can be correctly updated, and the analysis can know that the first `ptrtoint` does not make the pointer escape.

**Union workaround.** LLVM IR does not support unions, and C/C++ unions are typically lowered into character arrays or structs. It happens that when a C/C++ union contains a smaller type and a larger array, the union becomes a struct in the IR, with the smaller type as the first element and the array with the beginning “trimmed away” as the second element. Therefore, access to the beginning of the array becomes out-of-bound access to the second member inside the struct and will result in false positives. The preprocessing stage uses heuristics to identify all union types, and all `gep` from them will be replaced with a sequence of pointer typecast and `gep` such that the bound of the union is not narrowed to the member even if the static type is the same as a struct type.

### 4.4.1.2 Graph-based Analysis and Instrumentation

To accomplish the task of compiler instrumentation described in Section 3.4, the IR pass needs a static analysis phase that can answer the following two central questions:

1. What pointer use (dereferences) require checking; and

---

13Currently all structs with a name starting with “union.” is considered as a union.
2. What pointer source (object allocation) will have the value flow into checks.

For the first question, the analysis should eliminate as many checks statically as possible, and the instrumentation phase will insert checks before the rest of the pointer use sites. For the second question, the analysis should avoid instrumentation on objects or pointers that do not require promote-based checks. The instrumentation phase will instrument the remaining object allocations. The analysis phase can then find all pointer operations that require instrumentation for up-to-date metadata by identifying all possible dataflow from pointer sources to pointer uses that require checking, and all pointer operations along these value flow would need metadata update.

Because all the analysis above is based on the dataflow of pointer values, the core of the analysis and instrumentation are performed on a pointer dataflow graph. Therefore, after the preprocessing stage, the IR pass (1) loop over all the instructions in the IR and build the inter-procedural pointer dataflow graph, (2) run multiple dataflow analysis over the graph to solve certain properties on the graph nodes or edges to derive instrumentation decisions, and (3) perform the instrumentation on the graph. The rules for the static analysis to make instrumentation decisions are described in Section 4.4.1.4 below. The following explanation of the pointer dataflow graph will use the graph from Figure 4.3 as an example.

**Nodes.** A pointer dataflow graph contains three types of nodes listed below. Each node is displayed as a record in Figure 4.3; the first two rows of each record form a summary of the node that all the information is available from the source IR, and the (optional) third row is a summary of internal data and flags introduced by the IFP pass.

- **Pointer Def:** A node representing a definition of a pointer value. Any variable declarations, function arguments, etc. will create a pointer def node to represent the pointer value. These nodes are displayed in Figure 4.3 with the first line starting with “PtrDef”.

- **Pointer Use:** A node representing a use of a pointer value. A pointer dereference through load or store, a pointer argument to a call, etc. will create a pointer use node to represent a use of the pointer value. These nodes are displayed in the figure with the first line starting with “PtrUse”.

- **Hook:** A node representing LLVM IR constructs (e.g., instructions, global variable declarations, and function arguments) that backs associated pointer def or uses. Hook nodes and all their outgoing edges are displayed with dotted lines. When a hook node represents an instruction or a function argument, the first line of text on the node starts with the name of its parent function. For example, a hook node starting with @foo means that its parent function is foo().

Each pointer def or use node is associated with exactly one hook node, while each hook node can be associated with multiple def or use nodes. An edge drawn from hook nodes to pointer def or use represents their association.

In the code example in Figure 4.3, the local array declaration (array) in line 6 is lowered to an *alloca* instruction in LLVM IR that produces the pointer to the declared local array. The graph represents the *alloca* instruction using a hook node (the top-most node with dotted outline) and attaches a pointer def node under it. The left arm of the dataflow starting from that def node is an implementation artifact.\[14\] The left arm of the dataflow is an artifact of LLVM’s way of representing the lifetime of local objects. LLVM use lifetime intrinsics (llvm.lifetime.start() and llvm.lifetime.end()) to mark the live intervals of local objects to assist optimizations including the stack coloring. However, because the input pointer type for these intrinsics is i8* (which corresponds to void* in C/C++), the LLVM IR need to use a *bitcast* to cast the pointer to i8* before passing it to the
and is not related to the semantics from the source program. The right arm of the graph corresponding to the dataflow from the source code. The declared array is passed as a function argument to the call to foo(), and the pointer is dereferenced there. In LLVM IR, this procedure is expressed as follows:

1. Because the type of array is a pointer to int[4] while the call expects a pointer to int, the LLVM IR uses a gep to compute &array[0] first, which yields a pointer to int. The first use and the subsequent def at the right arm of the dataflow comes from this gep.

2. The output from the previous gep is used in the call to the function foo(). Because the call instruction takes the pointer input, a use node for that pointer use is created.

3. Inside the function foo(), the pointer ptr comes from the function argument. The argument is
Chapter 4. Implementation Details

represented with a hook node and the input pointer value is represented by a pointer def node under the hook node.

4. The argument pointer is then dereferenced by a store instruction that stores the value 0 to the memory pointed by the ptr from the argument. The address operand of the store is represented by a use node.

The formal rules for adding edges between the nodes are discussed next.

**Edges.** The edges between pointer def and use nodes (shorthanded as def and use) are the edges that later static analysis operates on. An edge from a def to a use means that the value represented by the def node is exactly the value being used in the use node. In other words, the value of the def node “flows into” the use node. Because the LLVM IR is in SSA form, each use will have one incoming edge from exactly one def node. An edge from a use to a def indicates one of the following situations:

- The def and use is associated with the same LLVM instruction that modifies the input pointer value from the use and produces the output pointer value to the def, and the semantics of the instruction ensures that in absence of spatial memory safety errors, the input and output pointers:
  1. should point to the same top-level object; and
  2. have the same value on poison bits.

- The edge represents a possible copy of the pointer value from the use to the def. This can happen when the def and uses are associated with the same hook (e.g., a phi or select instruction) or different hooks. For example, a function argument can pick up call arguments from multiple call sites, and the hook of the def (the function argument) is distinct from the hook(s) of the use(s) (all the call sites to the function).

The def may have multiple incoming edges from different uses, and the concrete value at the def will be one of the possible values from the incoming edge. A use may also have multiple outgoing edges to different defs, for example, a pointer return value may flow to any corresponding call sites. In the text below, if there is a path from def or use node A to a def or use node B, then node A is in the upstream direction of node B, and B is in the downstream direction of node A. If the path from A to B does not go through any other node, then node A is the immediate upstream of node B, and B is the immediate downstream of node A.

The graph representation allows the IR pass to efficiently attach data structures to nodes, and use a single implementation of the dataflow analysis algorithm to solve different properties necessary for instrumentation, which is explained next.

**Analysis on the graph.** The pointer dataflow graph is essentially a set of graph components where each component is a one-variable dataflow graph of a pointer. Same as conventional dataflow analysis, the IR pass uses forward and backward iterative dataflow analyses to solve the fixed-points of related properties, where a forward analysis would propagate and update information from upstream nodes to downstream nodes and a backward analysis would perform the operations from downstream nodes to upstream nodes.

---

15 A phi node is used in SSA forms to represent that a value can pick up different values depending on which basic block is the predecessor. For example, if there is an if-else block that assigns a variable with value v1 from the if branch and v2 from the else branch, then the LLVM IR would (1) contain code in the if and else branch to compute v1 and v2 correspondingly, and (2) create a phi node after the if-else block that takes v1 and v2 as inputs to represent the value of the assigned variable after the if-else block.

16 This instruction is equivalent to the ternary operator ((Condition)?(TrueValue):(FalseValue)) from C/C++.
upstream nodes. How the data is updated and propagated is encapsulated in the *meet operator* of the analysis. The iterative analysis is implemented by executing the meet operators for the node property in depth-first traversal, and the traversal stops when the data on all the nodes converge. The converged state is referred to as the fixed-point.

### 4.4.1.3 Graph building

The first task after the preprocessing stage is to build the pointer dataflow graph. The IR pass iterates over all the functions that are not excluded for instrumentation\(^\text{17}\) iterates over all the instructions, and creates nodes for all data pointer related operations. When a function call is encountered, the call site information is saved first, and after all the functions are processed, the use nodes of call arguments in callers are connected to the def nodes of function arguments in callees, and the use of return values are connected to the def node of call return values. The implementation currently does not support variadic functions; they are excluded from analysis and instrumentation, and the IR pass treats them as legacy code. Besides creating the nodes, the graph-building code also initializes the flags or data used for later analysis.

### 4.4.1.4 Graph-based Static Analysis

The static analysis phase needs to solve the following problems to guide the instrumentation:

- **Object instrumentation:** For each pointer def from a local/global variable declaration, whether any pointer to it would escape. IFP assumes that every escaping pointer can be *promoted* at a later time, so if one of the pointers does escape, then the variable allocation must be instrumented, and the *escaping* pointer need to carry the valid pointer tag to pass later promote checks.

- **Static check elimination:** For each pointer use, whether it is statically safe and does not need instrumentation, meaning that it is either (1) statically in bound in all possible scenarios, or (2) poisoned\(^\text{18}\).

- **Check placement:** Where to insert size checks to make sure each pointer use that is not statically safe is made safe at runtime by the checks. Either the pointer flowing into the use is made safe for dereferencing, in other words a check is performed somewhere upstream, or the pointer should be in IFPR form (i.e., carry a bound) and a size check is needed in place before the use.

The object instrumentation problem is solved using the following procedure. First, during graph building, all pointer uses that can make a pointer escape is marked. Then, this static analysis phase uses a backward dataflow analysis to identify and mark all pointer defs whose value may flow to an escaping use. When a def has multiple uses, the def is marked if any of the use it flows to is marked, and the same is applied when joining multiple defs to a single upstream use. Therefore, after the backward analysis converges, all marked pointer defs belonging to local or global variable declarations represent allocations needing instrumentation.

To perform the static check elimination, the IFP pass needs to (1) compute the *address offset* range for pointer defs that are guaranteed safe to access (referred to as *def range* below), (2) compute the

\(^{17}\)Clang is modified to support _attribute_((ifpoption(“ignore”))) function annotation, which disables the analysis and instrumentation on the annotated functions.

\(^{18}\)Dereferencing a poisoned pointer guarantees a crash, so if the pointer is poisoned in the “upstream” of value flow, then this use can still be left uninstrumented and no compromise can happen.
address offset range that each pointer use may access (referred to as *use range* below), and (3) try to prove that the use range always falls into the def range. The IFP pass uses both fixed-point iteration (which can be inter-procedural) and the LLVM Scalar Evolution (shorthanded as *SCEV*) analysis pass (which is intra-procedural) to compute the ranges. The overall algorithm for solving the problem is as follows:

1. During graph building, all defs directly from object allocations (including local/global variable declarations and dynamic memory allocations) have their initial def range set, and all uses that dereference a pointer will have their initial use range set. There are following special handlings involved:
   - **gep** that computes struct member address are specially handled in that (1) the pointer use is considered as accessing the entire struct; and (2) the pointer def has a def range of the entire struct. This effectively hoists all bound checks on a pointer to a struct member to the input pointer (to the base address of the struct).
   - While pointers from unknown origin (including pointers loaded from memory, cast from integer, or passed from another unknown function) are not safe to dereference directly, they are assumed to have a def range matching with the static type size, and a special flag is left so that if any pointer use has the bound check eliminated because of the def range provided, a size check is instrumented immediately after the pointer def.

2. In the first step of static analysis, all pointer defs that the graph-based analysis know how to compute the def range better than SCEV are identified. The value of these def nodes will be marked as *SCEVUnknown* so that SCEV will treat them as black boxes. These def nodes are referred to as SCEV *root* nodes in the text below. The IR pass considered the following node as SCEV root:
   - The def nodes of **gep** instructions that compute struct member address. If they are not marked as *SCEVUnknown*, the SCEV can “look through” the **gep** and express the range of downstream def/use to be based on a def at upstream of **gep** instruction, causing later analysis code to neglect the def range specified on the def node.
   - The def nodes of **phi** or **select** that selects one value from multiple independent values. Their def range can be computed as the intersection of the def range of all upstream values. **phi** nodes for pointer-type loop induction variables (where one input value is derived from the output value of **phi**) are still solved by SCEV.
   - The def nodes of call return values and function arguments. SCEV is intra-procedural and cannot solve the range across functions, while the inter-procedural analysis can.

3. After SCEV root nodes are found, their def ranges are computed using a forward analysis. When a def has multiple incoming values from different upstream defs, their intersection is taken as the result def range.

4. After all def ranges are solved, the IR pass iterates over all the uses and check if they always fall into the def range of the immediate def node. If the immediate def node for a use is not set as a SCEV root node (therefore may not have the def range computed), its def range is solved by (1) querying the SCEV pass to express the value of the immediate def node as a sum of one SCEV root
node and an offset expression from the SCEV root’s value, then (2) solving the bound of the offset expression to compute the def range of the immediate def node, and finally (3) check it against the use range. All uses not considered statically safe are marked.

After finding all uses that are not statically safe, the analysis code needs to determine where to insert size checks. The IR pass should try to hoist the size check to the upstream direction so that the live interval of pointer bounds can be shortened to reduce their spilling costs. The hoisting should also avoid introducing false positives, in other words, no size check should poison the value to a use that only requires a smaller check size.

To determine the instrument site for size checks, the IR pass performs a backward analysis to solve the check size attribute. The check size attribute on a def is the maximum safe size in bytes that each use can access without additional size checks, and the check size on a use is the minimum assured range that the use expects when the pointer value flows into the use. Therefore, size checks are instrumented (1) between a use and the upstream def when the check size of upstream def is smaller than the size on use; and (2) immediately after pointer defs when the def has non-zero check size and the semantics of the pointer source do not guarantee a safe-to-dereference size.

Initial condition. For all use nodes, the initial check size is the access size if the use is not statically safe, and zero if it is safe or it does not dereference the pointer. For all def nodes, the initial check size is the static type size for pointer sources that have a non-zero def range used in static check elimination.

Meet operator. When joining multiple uses to a single def, the check size of def is the smallest non-zero check sizes among the uses. When propagating from def to use, the behavior depends on the LLVM construct the def is associated with. If the def is from a function argument where the function can be called from uninstrumented code, or when the def is from a gep that performs pointer arithmetics, then the check size is not propagated as if the edges are ignored. In all other cases, the check size is copied to all upstream uses.

After the check size on each node converges and instrument sites are identified, the analysis code runs the last fixed-point algorithm to determine whether each node should carry an IFPR pointer or a GPR pointer. This ensures that the bound is preserved along the dataflow from the pointer origins (variable declarations, dynamic allocations, or promote-instrumented source) to the size checks.

4.4.1.5 Instrumentation

The instrumentation phase will take over the graph and modify the IR for the following tasks after the analysis phase is completed:

- Object allocation instrumentation. The instrumentation phase will (1) instrument local and global variable declarations that have pointers escaped, (2) instrument code to set up the pointer tags for escaping pointers, and (3) rewrite dynamic allocation function calls to the counterpart provided by the IFP runtime library. All these instrumentations should be performed on graph nodes.

- Pointer dataflow update. This includes (1) for all nodes that need the pointer to be in IFPR form, modify or replace the node’s backing construct to a counterpart that uses IFPR pointer types; (2) instrument pointer checks, including promote and size checks; (3) instrument subobject index update and bound narrowing for gep. The promote and size checks can be instrumented at any
code location between the def and uses (i.e., the checks are instrumented to edges), while the other two modifications must be performed on nodes.

For object allocation instrumentations, if the type information is available and it is a composite type, the instrumentation code will generate the layout table for the type of allocation and include the layout table metadata in the object metadata. The type information for local and global variables can be extracted from their declarations. The type of the object from a dynamic allocation is inferred from the use of the returned pointer.

This instrumentation phase is implemented with the following two steps:

1. Node update. The code loops over all def and use nodes, perform all node-based instrumentation, and update their declared pointer dataflow interface. Pointer uses with modified input interface (e.g., input type and which operand of which instruction) will take input from a temporary def, and pointer defs with modified out interface (e.g., output type and which instruction’s output) are left unused. After this step, the input and output interface of each graph node is finalized.

2. Edge update. This step re-connects all stale or missing edges among modified nodes in the dataflow graph. The IFP pass loops over all def and use nodes again, connect pointer defs to uses according to the updated interface, and instrument all necessary edge-based instrumentation (promote and size checks).

Algorithm 1: Pointer check instrumentation placement

Input: Location of pointer def $I_d$ and the locations of one or more use $\{I_{u_n}\}$
Result: The instrument location $I$ that is dominated by $I_d$ and dominating all $I_u \in \{I_{u_n}\}$

if $I_d$ is in the same basic block as one of $I_u \in \{I_{u_n}\}$ or one of the use is phi then

    return The first $I$ immediately before the first of: (1) the end of the basic block for $I_d$; (2) $I_u \in \{I_{u_n}\}$ in the same basic block as $I_d$;

Candidate Basic Block $BB_c \leftarrow$ common dominator for all BB of $I_u \in \{I_{u_n}\}$

if $BB_c$ in Loop $L_c$ then

    $L_t \leftarrow \emptyset$;

    if $I_d$ in Loop $L_d$ then

        $L_t \leftarrow$ common parent loop of $L_c$ and $L_d$;

    while $BB_c$ in Loop $L_c$, $L_c \neq L_d$, $L_c \neq \emptyset$ do

        if $L_c$ has preheader then

            $BB_c \leftarrow$ preheader of $L_c$;

        else if exists $BB_h$ that dominates the header of $L_c$ and is dominated by or equal to the basic block of $I_d$ then

            $BB_c \leftarrow BB_h$;

        else

            break;

    return The first $I$ immediately before the first of: (1) the end of basic block $BB_c$; (2) $I_u \in \{I_{u_n}\}$ inside $BB_c$;

When the pass find that a promote or ifpchk needs to be instrumented during edge updates, the instrumentation code use the heuristic in Algorithm 1 to determine the code location where the operation is instrumented. This heuristic tries to strike the balance between (1) sinking the check to avoid its cost
when the checked pointer is not used, and (2) execute the check as few times as possible. It first finds
the common dominator basic block for all uses, and then try to hoist the check out of unnecessary loops
if possible.

### 4.4.2 Backend Modification

The RISC-V backend of LLVM is run after all the IR pass to generate RISC-V instructions. It is
modified so that it (1) supports code generation for IFP operations, (2) implements the backward-
compatible calling convention for passing IFP across functions at call and return, and (3) performs
implementation-specific optimizations on IFP instrumentations.

**Calling Convention.** To pass IFPR across function calls and returns, the IR pass and the backend col-
laborate as follows. First, the IR pass annotate IFPR passed through call/return using new `argpromote`
and `argdemote` intrinsic functions. The formal function type is not changed for IFPR pointer passing;
instead, the formal type for arguments and return values are still GPR, but the recipient code can
pickup the passed IFPR through `argpromote`, and the sender of IFPR will pass it to `argdemote` and
its return value is used as the actual call argument or return value instead. Then, the backend looks
for `argpromote` and `argdemote` intrinsics to copy the shadow IFPR for data pointer arguments if one is
found, and otherwise resort back to copying the GPR when inserting register.

**Implicit Size Check Folding.** The IFP ISA extension states that when the GPR of a bound-checked
IFPR is used as the base address operand of load or store, the instruction would perform an implicit
size check. Therefore, if the result of an `ifpchki` is only used as the address operands of loads or stores
and it is possible to fold the size checks into these pointer uses, the backend optimization should try to
allocate the `input` IFPR of `ifpchki` to a bound-checked IFPR to eliminate the check. The backend uses
a two-stage process to implement this optimization. First, before the register allocation, the backend
identifies all `ifpchki` that can be folded into uses. Then, after the register allocation, the backend checks
whether the identified `ifpchki` have their input IFPRs being allocated in a bound-checked IFPR, and
for each `ifpchki` passing the check, the backend updates the users of output from `ifpchki` to take the
IFPR input of `ifpchki` instead, and remove the unused `ifpchki` left.

### 4.5 Software Modification

In this section, the changes to software support on the target RISC-V platform are listed.

**Operating System.** The CVA6 SDK repository uses its own Linux fork [33], which is based on kernel
version 5.1.0. The kernel is modified so that (1) all bound registers and CSRs are saved and restored
when entering and leaving user mode, and (2) pointer checks on user-level pointers and data copy from/to
user-level memory are modified to ignore the pointer tag except the poison bits.

**Runtime Library.** The In-Fat Pointer implementation introduces a runtime library that is linked
to all IFP-protected user-level programs. This library populates CSR values at application start, and

---

19 This is an arbitrary implementation decision made to avoid rewriting LLVM IR everywhere that must be consistent
with the function type.
provide dynamic memory allocation functions that instrumented code calls to. To evaluate all of the metadata schemes, there are two dynamic memory allocators implemented in the runtime library: (1) a wrapped allocator built on top of libc's `malloc()` and `free()` which uses Local Offset scheme if possible and falls back to Global Table scheme; and (2) a subheap allocator implemented as a pool allocator on top of a buddy allocator, which uses Subheap scheme. The compiler instrumentation will rewrite calls to allocator functions like `malloc()` and `free()` to the counterpart from the selected allocator.
Chapter 5

Evaluation

This section presents the evaluation on the prototype implementation of In-Fat Pointer. First of all, in Section 5.1, In-Fat Pointer is evaluated functionally by testing whether In-Fat Pointer can successfully catch spatial memory errors using the NIST Juliet test suite \[31\]. Then, Section 5.2 evaluates the performance of promote with a microbenchmark. Section 5.3 measures the runtime and memory overhead imposed by In-Fat Pointer on four applications. Finally, Section 5.4 estimates the hardware area overhead from the prototype implementation.

**Experimental setup.** For all the experiments, Xilinx Vivado 2018.2 is used to synthesize the CVA6 SoC with the modified CVA6 processor on a Digilent Genesys 2 FPGA development board, which has a Kintex-7 XC7K325T-2FFG900C FPGA device and 1GB of DDR3 RAM. The processor runs at a fixed frequency of 50MHz.

**Runtime library support for evaluation.** As mentioned in Section 4.5, there are two allocators implemented in the runtime library: a wrapped allocator built on top of libc’s allocator, and a subheap allocator implemented with mmap(). The runtime library also collects additional performance statistics including readings from hardware performance counters. The runtime library is statically linked into the program with --whole-archive linker option to ensure the execution of stat collection code even when no instrumentation is performed and none of the symbols from the runtime library is referenced from the program body. Because of the current implementation of subheap allocator, all experiments are run with /proc/sys/vm/overcommit_memory set to 1.

### 5.1 Functional Evaluation

The NIST Juliet test suite 1.3 for C/C++ \[31\] is ran to evaluate the correctness of In-Fat Pointer. 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. Programs under the following category are chosen: buffer overflow (stack-based and heap-based), underwrite, overread, and underread. These programs feature out-of-bound memory access that In-Fat Pointer intend to catch. Because In-Fat Pointer sets the poison bits whenever a pointer check fails, the crash of instrumented programs because of poisoned pointer dereference is the indication that In-Fat
Chapter 5. Evaluation

Pointer detects spatial memory errors successfully. Among the total of 16,572 test cases in the selected categories, all 6,484 C++ test cases are excluded because the prototype currently does not support C++. For the remaining 10,088 test cases, those that have external dependencies (e.g., requiring network connections, files, or input from stdin) and those that only run on Windows are excluded, leaving 9,328 test cases.

The test cases are further filtered down and some of them are modified as follows. Among the 9,328 initial test cases, 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 the random number generation functions are modified 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. Then, 2,672 test cases that satisfy one of the following conditions are also removed: (1) the test program has the only memory error triggered in (uninstrumented) libc functions (e.g., strcpy()); (2) the crash requires specific (non-zero) initial value from uninitialized memory; (3) the test case only trigger the memory error on a 32-bit platform. These test cases are suitable for evaluating static analysis tools instead of dynamic checkers like In-Fat Pointer. After the adjustment, among the 6,656 test cases remaining, 1,084 of them have their spatial error optimized away or becomes benign by the compiler at -O1 optimization level, which is the minimum level that In-Fat Pointer compiler instrumentation pass can run on, therefore these test cases are excluded as well, leaving 5,572 valid test cases for In-Fat Pointer. All 5,572 remaining test cases with both vulnerable and non-vulnerable code are then tested on the In-Fat Pointer prototype, and In-Fat Pointer successfully crashes the vulnerable code while letting all non-vulnerable code work, thus having full accuracy with no false positives on these selected test cases.

5.2 Microbenchmark

promote instructions are the most costly instructions introduced by In-Fat Pointer because they require one or more memory accesses and arithmetic operations for fetching and processing in-memory metadata. In this section, the cost of each promote instructions is estimated using a microbenchmark. The goal is to find the latency of a promote instruction when the following variables are changed:

- The object metadata scheme used by the pointer and the object;
- Whether the layout table is used or not; and
- The number of layout table element fetching needed for subobject bound narrowing.

To evaluate the performance impact from the first two factors, the microbenchmark used in this section keeps executing promote instructions on pointers to the following object:

1. an uninstrumented object as the baseline
2. an object using the global table scheme
3. an object using the local offset scheme, and the object metadata includes the layout table metadata

1 The prototype implementation currently assumes that Mem2Reg optimization pass is run, which is enabled at -O1.
2 Latency is the number of cycles needed to finish the execution of a single instruction. Because the IFP unit is not pipelined in the current implementation, latency can fully describe the performance characteristics of promote.
3 The global table scheme implementation does not support layout tables.
4. an object using the local offset scheme, but no layout table is used

5. an object using the subheap scheme, and the layout table is used

6. an object using the subheap scheme, but no layout table is used

To evaluate the impact of the number of layout table elements needed, the microbenchmark uses the type from Figure 3.10 for all objects, where the layout table has 6 elements that:

- **promote** on a pointer with subobject index 0 do not need to access the layout table, as usual.
- **promote** on a pointer with subobject index 1, 2, and 5 will need to fetch one element from the layout table (which is the element directly pointed by the subobject index).
- **promote** on a pointer with subobject index 3 and 4 will need to fetch two elements from the layout table (which are (1) element 2, and (2) the element pointed by the subobject index).

In each run of the microbenchmark, the code will prepare one pointer for each combination of objects and valid subobject indices, and the address part of the pointers are adjusted so that each of them is an in-bound pointer for the corresponding subobject. For each pointer, the microbenchmark uses a tight loop with three instructions: a decrement of the loop counter, a **promote**, and a loop-exit branch. The code reads RISC-V’s cycle performance counter immediately before and after the loop to get the total number of cycles elapsed for the loop.

![Figure 5.1: Promotion performance microbenchmark](image)

Figure 5.1 shows the average cycle count for each loop iteration for each combination of objects and subobject indices. The primary series along the horizontal direction is the object, and the secondary series represent the subobject indices. The subobject index increases from 0 to 5 when going from left to right on the figure. Each loop runs 1,000,000 iterations, and the measured cycle count is divided by the number of loop iterations to compute the average cycle count. The result is averaged from 10 runs of the microbenchmark. Because each loop body consists of 3 instructions, and the CVA6 processor is single-issue, the cycle count per loop is always at least 3. In practice, all the computed average cycle counts are slightly higher than the real cycle counts in integers because of intermittent OS activities. All the code and metadata used by the loops will fit in the cache. There are the following two notes from further analysis on the microbenchmark:

4 These conclusions are derived from observing the execution of the microbenchmark on the cycle-accurate emulator generated by Verilator.
The performance bottleneck in the baseline scenario is at the front end of the processor pipeline. The abnormal 4-cycle iteration for subobject index 0 and 4 are caused by the instruction memory layout as an implementation artifact. For these two cases, the loop-exit branch instruction is at the end of a cache block in the instruction cache, and the cache needs an extra cycle to cancel the request to the next block when the instruction fetch logic realizes that there is a branch that is predicted to jump backward. This problem does not occur on other objects with metadata because the promote operation is the bottleneck and the latency at the front end is hidden.

The promote operation becomes the performance bottleneck in the loop body when it takes a pointer from an instrumented object. In these cases (except in baseline where the promote is single-cycle), the latency of promote equals to the average cycle count minus one cycle. The branch prediction and speculative execution do not completely hide the latency because the register renaming is not enabled and the issue logic needs an extra cycle to resolve the write-after-write hazard on the destination register of promote.

As mentioned in Section 4.3.2, the memory load from the IFP unit is not pipelined, and each load has 3 cycles of latency and can at most load 16 bytes of metadata in each request. Therefore the empirical results from Figure 5.1 match well with the implementation, which is first concluded as follows and then explained in the rest of this section:

- A promote to a pointer using the global table scheme takes 4 cycles, with 1 cycle for fixed overhead and 3 cycles for a metadata load;
- A promote to a pointer using the local offset scheme takes 5 cycles, with 2 cycles for the fixed overhead and 3 cycles for a metadata load;
- A promote to a pointer using the subheap scheme takes 8 cycles, with 2 cycles for the fixed overhead and 6 cycles for two metadata loads.
- The cost of a promote on layout table is (1) 3 cycles per element fetch and (2) one (or more) cycles for each bound narrowing operations from the layout table.

Performance impacts from object metadata schemes. The cost of promote from each metadata schemes can be found by computing the latency of promoting pointers to objects instrumented with that scheme but with no layout table. Because no subobject bound narrowing is present due to lack of layout table, the cycle count stays the same for different subobject indices.

All the handling logic for object metadata schemes need to make at least one metadata load request, which has 3 cycles of latency. The IFP unit also has a one-cycle overhead to handle the instructions and assemble the results. Therefore, all promotes on an object with metadata need at least 4 cycles in the current implementation. Besides the cycles spent on metadata load, the extra fixed one cycle overhead for the local offset scheme and the subheap scheme comes from the use of MAC; after the metadata is fetched, the logic needs to wait until the MAC verification finishes. The subheap scheme is currently the slowest because it both uses the MAC and needs to fetch 32 bytes of metadata which is larger than the metadata of the other two schemes. Although its 32-byte metadata is contiguous in memory, the IFP unit still has to make two un-pipelined requests to fetch the metadata. For future works, to improve the latency equals to the average cycle count minus one, as explained in the earlier notes.
performance of promote, one may optimize the implementation to pipeline the memory access or reduce its latency and improve the design of the subheap scheme to shrink the metadata size.

**Performance impacts from layout table usage.** The performance impact from the layout table and the subobject bound narrowing can be found by comparing the latency of promoting pointers from objects instrumented with the same metadata scheme but one with the layout table is enabled and the other disabled. As shown in Figure 5.1 except that the local offset scheme implementation can hide one cycle of latency from layout table operations, the layout table incurs the same overhead on local offset scheme and subheap scheme for the same subobject index as expected, therefore the discussion below does not distinguish the “base” object metadata schemes that support the layout table.

As shown from the figure, the performance overhead from the layout table is highly dependent on (1) the number of layout table elements fetched, and (2) the number of bound narrowing operations replayed on the object bound. In the current implementation, each layout table element fetch takes 3 cycles, which is the same as other metadata loads. Then, as described in Section 4.3.2.2, each fetched layout table element will be “decoded” into one or both of (1) a single-cycle struct member bound narrowing in which only additions and subtractions are involved, and (2) a multi-cycle array element bound narrowing where there is a division on the array element size. For the example in Figure 5.1 the array element size for S.array (element #2) is 8, which is a power-of-two size, therefore the array element bound narrowing is also single-cycle. Therefore, for each non-zero subobject index, the layout table imposes:

- a fixed 4-cycle latency when the subobject index is 1, 2, or 5; 3 cycles come from the layout table element fetching and 1 cycle from the struct member bound narrowing operation.
- a fixed 9-cycle latency when the subobject index is 3 or 4. In addition to the bound narrowing to #2, #3 and #4 need to (1) fetch one extra layout table element (3 cycles), (2) one array element bound narrowing from the entire S.array (#2) to the bound of the array element (1 cycle), and (3) one struct member bound narrowing from the array element to the struct member (1 cycle).

As shown from the previous discussions, the variance of imposed performance overhead from the layout table is large, and the variance depends on many factors including array element type size and the type (or subobject) hierarchy in the layout table. Therefore, one can expect that the evaluated impact is different when evaluating the layout table on real programs.

### 5.3 Performance Evaluation

The evaluation in this section aims to answer the following questions: (1) what is the overall runtime and memory overhead imposed by In-Fat Pointer; and (2) where is the imposed runtime overhead comes from. In-Fat Pointer can impose runtime overhead due to the following reasons:

- Additional pointer checking and metadata update code being instrumented, which can be further broken down into:
  - promote instructions that are few but are slower than all other instructions because of the memory loads and complex logic

---

6 The divider can perform divisions on power-of-two divisors in a single cycle.
– slightly more store instructions for object metadata maintenance
– other arithmetic-only instructions that increase the dynamic instruction count

• Additional register pressure because of instrumented checks and incomplete optimization support in the implementation\(^7\)
• Additional cache pressure caused by accesses to in-memory metadata (for data cache) and additional instructions (for instruction cache)

Therefore, to understand the contribution of each factor of overhead, the following factors are measured or estimated:

• The slower performance of \texttt{promote} instruction
• The increase in register pressure and memory working set size
• The increase in instruction count

The following programs are selected to evaluate the runtime and memory overhead: (1) bzip2\(^{39}\) 1.0.8 compressing its own source code tarball, (2) 458.sjeng from SPEC2006 benchmark\(^{15}\) (short-handed as \texttt{sjeng} in the text below) with reduced search depth and runtime, (3) CoreMark\(^{16}\) with fixed 1000 iterations, and (4) WolfCrypt’s Diffie–Hellman benchmark\(^{43}\) (short-handed as \texttt{wolfcrypt-dh}). CoreMark is a popular benchmark for embedded systems, and it tests the performance of matrices and linked lists, which are expected to have the performance results generalizable to real programs that use arrays and other pointer-rich data structures. The remaining programs are selected to represent common workloads including compression/decompression, artificial intelligence, and encryption/decryption. Each program is run for at least three iterations and the average is taken when gathering the statistics. The \texttt{time -v} command is used to execute the program, and the elapsed wall time and maximum resident set size is used for runtime\(^8\) and memory usage.

To evaluate all of the metadata schemes, each program is compiled into three variants: (1) uninstrumented baseline, (2) instrumented version using wrapped allocator, and (3) instrumented version using subheap allocator, and their runtime and memory consumption is compared to compute the overhead data. To investigate the contributing factors of the runtime overhead, the experiment is extended to (1) use hardware performance counters to measure the change in total dynamic instruction count and executed load/store instruction count; (2) create the runtime library variant that enables a \texttt{no-promote} mode in hardware introduced below, and run the instrumented programs linked against this runtime library variant in addition to the normal one.

The \texttt{no-promote} hardware mode helps to isolate the performance impact caused by \texttt{promote} instructions from all the other factors. In the no-promote mode, whenever a \texttt{promote} instruction is executed, the hardware populates the IFPR with an infinite bound in a single cycle and does not make any memory access, making it equivalent (in cost) to a register move. Therefore the no-promote version of instrumented programs will have the same memory footprint and instruction count (subject to small

\(^7\)The current compiler implementation of In-Fat Pointer do not support rematerializing a pointer with pointer tag from another pointer to the same object but different address. This includes pointers to local variables; they cannot be rematerialized from stack pointers. Instrumented checks can introduce multiple versions of the same pointer. For example, a single pointer may have an unchecked version for pointer comparison, and one or more checked versions (differentiated by check sizes) for other uses.

\(^8\)Note that the runtime overhead of CoreMark is retrieved from CoreMark’s own reporting output instead of \texttt{time -v} because it runs the shortest time and the start-up overhead becomes non-negligible when \texttt{time -v} is used.
variations) but reduced runtime overhead, and the reduction in runtime is then measured to compute the contribution of promote instruction in the overall runtime overhead.

Note that all program versions run on the same modified OS and are linked to the IFP runtime library, including the uninstrumented baseline programs. Additional experiments show that whether the uninstrumented program runs on a modified OS or vanilla OS has a negligible impact on runtime or memory consumption, and linking to the IFP runtime library incurs negligible runtime overhead but may incur a small fixed memory overhead. The discussion on this fixed memory overhead is in Section 5.3.2 below.

Note that according to statistics from the hardware performance counters, no promote instruction accessed the layout table during the evaluations. In other words, the performance result presented below does not show the performance impact of layout tables. This is caused by the following two factors:

- The layout table is only accessed when promoting an interior pointer, which are pointers pointing to a subobject inside the allocation. However, the evaluated programs usually either simply allocate large arrays with no subobjects, or only pointers to the beginning of objects are saved and loaded from memory (in which case the IFPR cannot be passed and a promote is needed). In-Fat Pointer provides subobject-granularity protection even when layout tables are not accessed in these cases, because the pointer bound is already exact after promote, and any subsequent pointer arithmetic that triggers a bound narrowing will let the compiler narrow the object bound to the subobject immediately.

- For dynamic memory allocations, three out of four benchmarks (except 458.sjeng) do not use libc’s allocators directly. bzip2 uses function pointers to support replacing allocators. wolfcrypt uses its own memory allocation wrappers to support both allocator callbacks and memory allocation tracing and debugging. These behaviors stop the compiler from inferring the use type of dynamically allocated objects and therefore no layout tables for these objects are created.

The author acknowledges that it is unclear from the current evaluation which factors contribute more to the absence of layout table access. The support for custom allocation wrappers is left as future works.

### 5.3.1 Runtime Overhead

In this subsection, the experimental results are presented and explained first, then the analysis and discussion on the overhead distribution are elaborated.

![Figure 5.2: Runtime overhead on evaluated applications.](image-url)
Table 5.1: Average runtime (in seconds) of evaluated applications

<table>
<thead>
<tr>
<th></th>
<th>wolfcrypt-dh</th>
<th>sjeng</th>
<th>coremark</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>93.15</td>
<td>384.23</td>
<td>16.55</td>
<td>369.73</td>
</tr>
<tr>
<td>subheap</td>
<td>109.13</td>
<td>446.78</td>
<td>20.37</td>
<td>404.30</td>
</tr>
<tr>
<td>wrapped</td>
<td>107.77</td>
<td>444.40</td>
<td>18.89</td>
<td>402.63</td>
</tr>
<tr>
<td>subheap (no promote)</td>
<td>108.18</td>
<td>444.70</td>
<td>18.11</td>
<td>402.83</td>
</tr>
<tr>
<td>wrapped (no promote)</td>
<td>107.78</td>
<td>444.44</td>
<td>18.14</td>
<td>401.86</td>
</tr>
</tbody>
</table>

Figure 5.2 shows the runtime overhead of In-Fat Pointer on the selected programs, and Table 5.1 is the original runtime data. The geo-mean runtime overhead for the wrapped allocator variant is 13.57% and that of the subheap allocator is 16.37%. To better understand the decomposition of the runtime, the dynamic instruction count statistics are presented next.

Table 5.2: Dynamic instruction counts and composition on evaluated applications

<table>
<thead>
<tr>
<th></th>
<th>Total Instr</th>
<th>Load</th>
<th>Store</th>
<th>IFP Total</th>
<th>Promote</th>
<th>Bound L/S</th>
</tr>
</thead>
<tbody>
<tr>
<td>wolfcrypt-dh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>baseline</td>
<td>3.257 × 10⁹</td>
<td>6.386 × 10⁸</td>
<td>2.576 × 10⁸</td>
<td>2.284 × 10⁴</td>
<td>0</td>
<td>2.284 × 10⁴</td>
</tr>
<tr>
<td>subheap</td>
<td>3.742 × 10⁹</td>
<td>6.495 × 10⁸</td>
<td>2.660 × 10⁸</td>
<td>9.020 × 10⁵</td>
<td>7.909  × 10⁶</td>
<td></td>
</tr>
<tr>
<td>wrapped</td>
<td>3.697 × 10⁹</td>
<td>6.415 × 10⁸</td>
<td>2.626 × 10⁸</td>
<td>8.919  × 10⁵</td>
<td>7.773  × 10⁶</td>
<td></td>
</tr>
<tr>
<td>sjeng</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>baseline</td>
<td>5.743 × 10⁹</td>
<td>8.001 × 10⁸</td>
<td>7.880 × 10⁸</td>
<td>9.324 × 10⁵</td>
<td>0</td>
<td>9.324 × 10⁵</td>
</tr>
<tr>
<td>subheap</td>
<td>6.089 × 10⁹</td>
<td>8.718 × 10⁸</td>
<td>8.639 × 10⁸</td>
<td>7.062 × 10⁵</td>
<td>2.263 × 10⁶</td>
<td>1.044 × 10⁸</td>
</tr>
<tr>
<td>wrapped</td>
<td>6.088 × 10⁹</td>
<td>8.714 × 10⁸</td>
<td>8.636 × 10⁸</td>
<td>7.057 × 10⁵</td>
<td>2.176 × 10⁶</td>
<td>1.044 × 10⁸</td>
</tr>
<tr>
<td>coremark</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>baseline</td>
<td>4.556 × 10⁸</td>
<td>6.732 × 10⁸</td>
<td>2.756 × 10⁷</td>
<td>3.992 × 10⁵</td>
<td>0</td>
<td>3.992 × 10⁵</td>
</tr>
<tr>
<td>subheap</td>
<td>4.927 × 10⁸</td>
<td>7.142 × 10⁸</td>
<td>3.153 × 10⁷</td>
<td>6.392 × 10⁷</td>
<td>2.142 × 10⁷</td>
<td>3.026 × 10⁶</td>
</tr>
<tr>
<td>wrapped</td>
<td>4.919 × 10⁸</td>
<td>7.120 × 10⁸</td>
<td>3.136 × 10⁷</td>
<td>6.392 × 10⁷</td>
<td>2.142 × 10⁷</td>
<td>3.025 × 10⁶</td>
</tr>
<tr>
<td>bzip2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>baseline</td>
<td>8.344 × 10⁹</td>
<td>1.565 × 10⁹</td>
<td>2.949 × 10⁸</td>
<td>9.035 × 10⁴</td>
<td>0</td>
<td>9.035 × 10⁴</td>
</tr>
<tr>
<td>subheap</td>
<td>9.096 × 10⁹</td>
<td>1.592 × 10⁹</td>
<td>3.001 × 10⁸</td>
<td>2.163 × 10⁹</td>
<td>1.221 × 10⁷</td>
<td>1.706 × 10⁷</td>
</tr>
<tr>
<td>wrapped</td>
<td>9.097 × 10⁹</td>
<td>1.592 × 10⁹</td>
<td>3.001 × 10⁸</td>
<td>2.163 × 10⁹</td>
<td>1.221 × 10⁷</td>
<td>1.706 × 10⁷</td>
</tr>
</tbody>
</table>

Table 5.2 shows the dynamic instruction count gathered from the hardware performance counters by the runtime library. The “IFP Total” column counts how many instructions introduced by In-Fat Pointer is executed, including promote, IFPR bound load/store instructions (which is the “Bound L/S” column in Table 5.2), and other IFP instructions that do not access memory and is not present in original RISCV instruction set. The “Total Instr” column includes the count for all instructions, including IFP.
instructions. The “Load” and “Store” column counts the total number load and store instructions being executed, including IFPR bound load/storeinstructions. In other words, a `ldbnd` instruction will be counted in both the “Bound L/S” column and “Load” column, in addition to “Total Instr” and “IFP Total” column. All the percentage values below the absolute instruction counts use the total instruction count (“Total Instr”) of the baseline as the reference. Note that all these instruction counts also include kernel activities in addition to the user-level code, therefore the baseline versions of the evaluated programs also have the execution of bound load and store instructions counted during context switches.

<table>
<thead>
<tr>
<th></th>
<th>GPR Load/Store</th>
<th>Arithmetic, Branch, and Misc Instructions</th>
<th>IFP Instr</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>baseline</strong></td>
<td>22.29%</td>
<td>77.71%</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>wrapped</strong></td>
<td>22.67%</td>
<td>61.35%</td>
<td>25.20%</td>
</tr>
<tr>
<td><strong>subheap</strong></td>
<td>22.47%</td>
<td>61.35%</td>
<td>25.20%</td>
</tr>
<tr>
<td><strong>coremark</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>baseline</strong></td>
<td>20.82%</td>
<td>79.17%</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>wrapped</strong></td>
<td>21.85%</td>
<td>72.09%</td>
<td>14.03%</td>
</tr>
<tr>
<td><strong>subheap</strong></td>
<td>21.93%</td>
<td>72.19%</td>
<td>14.03%</td>
</tr>
<tr>
<td><strong>sjeng</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>baseline</strong></td>
<td>27.65%</td>
<td>72.35%</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>wrapped</strong></td>
<td>28.39%</td>
<td>65.32%</td>
<td>12.29%</td>
</tr>
<tr>
<td><strong>subheap</strong></td>
<td>28.40%</td>
<td>65.32%</td>
<td>12.30%</td>
</tr>
<tr>
<td><strong>wolf/crypt-dh</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>baseline</strong></td>
<td>27.52%</td>
<td>72.48%</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>wrapped</strong></td>
<td>27.52%</td>
<td>58.61%</td>
<td>27.70%</td>
</tr>
<tr>
<td><strong>subheap</strong></td>
<td>27.87%</td>
<td>59.33%</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: Dynamic instruction count composition on evaluated applications. All numbers are percentage over baseline total instruction counts.

To better visualize the data from Table 5.2, a chart showing the overall instruction distribution is in Figure 5.3 shows the overall distribution of dynamic instructions executed. Instruction counts from Table 5.2 are normalized against the total number of instructions in the baseline version of each program, therefore the cumulative value is the percentage increase in dynamic instruction count for each variant compared against the baseline. The IFP instruction component from Figure 5.3 is further broken down in Figure 5.4. Note that because IFP replaced pointer arithmetic instructions for pointers (that are subject to checks) with new IFP instructions, the sum of dynamic instructions from the original RISC-V ISA is less than 100%; their reduction results from the increase in dynamic counts of IFP instructions.

5.3.1.1 Runtime overhead from promote instructions

The performance of `promote` instruction can be compared by (1) comparing the promotion-enabled version and no-promote version using the same allocator; and (2) comparing two promotion-enabled version using different allocators. Besides the cost of `promote` instructions, the dynamic memory allocator performance could also contribute to the overhead. However, as shown in Figure 5.2, the runtime over-
Chapter 5. Evaluation

Figure 5.4: IFP instruction dynamic count composition on evaluated applications.

head between subheap allocator variant and wrapped allocator variant in no-promote mode is less than 0.5%, therefore one can consider that the cost of promote is the dominating factor for the performance difference for both comparison methods.

As shown in Figure 5.2, the difference in runtime overhead between the no-promote versions and the corresponding original versions are approximately less than 1% for three out of four benchmarks, with an exception on coremark where the difference is 13.68% for the subheap allocator variant and 4.52% for the wrapped allocator variant. According to Table 5.2, 4.347% of total executed dynamic instructions are promote instructions in instrumented coremark, while all remaining benchmark has < 0.2% promote instructions, which explains the difference in runtime between promotion-enabled an no-promote versions for these benchmarks. Therefore, 4.347% of promote instruction causes an additional (comparing with a no-op) runtime overhead of approximately 13.68% for the subheap scheme and 4.52% for the other two scheme because of the memory accesses and the promotion logic. Note that the overhead from coremark may be close to the upper bound of performance overhead imposed by In-Fat Pointer; according to Table 5.2 the ratio between regular load and promote is approximately 3.19 : 1, meaning that one load among every 3.19 load is loading a pointer that gets promoted later on. No other evaluated programs have this ratio smaller than 100 : 1.

5.3.1.2 Runtime overhead from other factors

Beside the promote instruction, the increased instruction count is the other significant factor in the runtime overhead. As shown from Figure 5.3 and Table 5.2 the dynamic instruction count increase ranges from 6.0% (for sjeng’s wrapped allocator variant) to 13.5% (for wolfcrypt-dh’s subheap allocator variant). This instruction count increase is roughly in-line with the runtime overhead increase for both wolfcrypt-dh and bzip2.

The missing gap between the runtime overhead (≈ 16%) and dynamic instruction count increase (≈ 6%) on sjeng can be explained by the increased register pressure and cache footprint. sjeng has the largest working set size among all evaluated programs, and is sensitive to changes in cache...
footprint, especially when the cache is as small as 32KB. Firstly, as shown from Table 5.2 among the
6% instruction increases, about 43% are loads and stores, which is likely caused by register spilling.
Next, from the hardware performance counters, the subheap allocator variant suffers 52.11% more data
cache misses and the wrapped allocator variant suffers 47.32% more data cache misses, and both of
them have about 37.08% more instruction cache misses. This increase is significantly higher than that
from other benchmarks; wolfcrypt-dh has the increase in data cache miss of approximately 22% for
both subheap allocator variant and wrapped allocator variant, and the rest two benchmarks have that
increase close to or much lower than the runtime overhead.

To reduce the overhead from instruction count increase and register pressure, the implementation
of In-Fat Pointer needs to (1) improve the optimization support for IFP constructs, and (2) explore
more aggressive compiler optimization techniques, including whole-program or link-time optimization.
These techniques may help the compiler to prove that more pointers are statically safe and therefore
the compiler can instrument fewer pointers and objects. In the current implementation, the graph-based
analysis and instrumentation do not exploit control flow information, and the (edge-based) pointer check
instrumentation placement is made by a heuristic. One of the possible future work is to combine
the control flow information into the dataflow graph to (1) statically optimize away more checks (e.g.,
the second iteration over an array should be safe after the first iteration on the same array), and (2) place
the checks and pointer tag updates using better algorithms.

5.3.2 Memory Overhead

In this subsection, the memory usage of selected applications is analyzed. The results are from the
maximum resident set size reported by time -v command. There is only random fluctuation in memory
overhead for instrumented programs between the case when promotions are disabled and when they are
enabled, so only the result when promotions are enabled is present.

![Figure 5.5: Memory overhead on evaluated applications.](image)

Table 5.3: Average memory consumption (in KB) of evaluated applications.

<table>
<thead>
<tr>
<th></th>
<th>wolfcrypt-dh</th>
<th>sjeng</th>
<th>coremark</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>13024</td>
<td>719440</td>
<td>4592</td>
<td>32944</td>
</tr>
<tr>
<td>subheap</td>
<td>15253.33</td>
<td>719744</td>
<td>4848</td>
<td>34368</td>
</tr>
<tr>
<td>wrapped</td>
<td>13248</td>
<td>719632</td>
<td>4592</td>
<td>32992</td>
</tr>
</tbody>
</table>

Figure 5.5 and Table 5.3 shows the peak memory overhead of listed programs for both the wrapped
allocator version and the subheap allocator version. First of all, the wrapped allocator version incurs less
than 2% additional memory overhead in all benchmarks, and the overhead drops when the baseline uses more memory. The overhead mainly comprises the increased code size, register pressure (and therefore stack size), as well as additional object metadata. Comparing with previous fat pointer schemes imposing $2 \times$ memory overhead\cite{32} which can scale linearly with the program’s memory consumption, In-Fat Pointer’s object metadata approach is more memory efficient, and the overhead scales sub-linearly with the total memory usage. For the subheap allocator version, the larger memory overhead comparing with the wrapped allocator version mainly comes from the fragmentation from the pre-allocation of blocks in subheap allocator implementation, which is still bounded and becomes negligible when the program uses more memory.

Note that all memory usage is collected when the IFP runtime library is statically linked into the program, which incurs about 200KB of fixed memory overhead for sjeng and coremark but near-zero overhead for wolfcrypt-dh and bzip2. First of all, if the baseline is changed to the program version without the IFP library linked in, then only the memory overhead of coremark will increase to about 13.5% and 7.5% for the subheap allocator variant and wrapped allocator variant respectively because the baseline use the least amount of memory, and all other benchmarks have the change less than 1%. The overall memory overhead is still less than 18% in all cases. Secondly, this fixed overhead is an artifact of the (memory-inefficient) implementation of the IFP runtime library instead of the fundamental memory usage increase from In-Fat Pointer instrumentation. The size of the library can be reduced by simply removing statistics gathering code, drop the \texttt{--whole-archive} linker option, or change the global table from a global array to a dynamically allocated array, for example.

### 5.4 Hardware Overhead

To evaluate the hardware cost of the prototype implementation, the resource utilization and timing reports from Vivado is analyzed. All hardware debug modules introduced by IFP implementation were disabled before the synthesis when generating the reports. Only the usage inside the CVA6 module is reported, which excludes all the other components from the SoC. While the exact resource usage varies on each synthesis, the variation is usually less than 0.1%, so only the numbers of a single synthesis run are presented. After the debugging logic is disabled, the modified CVA6 processor uses 59,261 LUTs and 32,545 FFs in total. Comparing with the vanilla CVA6 that uses 37,088 LUTs and 21,993 FFs, there is a 59.78% increase in LUTs and 47.98% in FFs. Correspondingly, the critical path delay increased from 16.420ns to 18.590ns where all the increase is contributed by the extra routing delay\footnote{The critical path delay breakdown is available from the Vivado-generated synthesis reports.}. 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 5.6 shows the LUT usage decomposition at each stage of CVA6. While the usage breakdown is not fully precise because the synthesis tool can perform cross-boundary optimizations, it still provides insight on overhead distribution. The largest contributing factor of the LUT usage increase comes from the new IFP unit in the execute stage (31.57%), followed by the change to the load-store unit (15.36%), then the scoreboard (14.65%). The IFP unit and changes to the load-store unit are elaborated in Section 4.3 and their high contribution to the LUT usage increase is expected. The growth in the scoreboard likely comes from the increased writeback port count and the forwarding logic for pointer bounds. The vanilla CVA6 core features a scoreboard with eight entries and four writeback ports that
have full operand forwarding paths for both integer and floating-point registers. The IFP implementation added one writeback port for the IFP unit and replicated the forwarding logic for bounds. The remaining usage increase in the issue stage comes from the additional bound registers. The small increase in the cache subsystem comes from (1) one more read port to the load unit, and (2) the support for parallel loads on the same cache block.

Figure 5.7 shows the LUT usage decomposition inside the IFP unit. The layout table walker is the largest module in the IFP unit that uses 3059 LUTs in total, and its bound narrowing module alone is using 1476 (48.25%) of layout table walker’s LUTs. The MAC computation unit is the second-largest module in the IFP unit, and it is expected that a production-quality MAC computation unit will take more overhead. Comparing with the global table scheme, the more complex object bound computation and the use of MAC pushes up the overhead of the local offset scheme and that of the subheap scheme, but their LUT usage is lower than the MAC computation and the layout table walker.
Chapter 6

Limitations

There are several limitations of the current design and implementation of In-Fat Pointer. Some limitations are inherited from the tagged-pointer scheme, and some are the artifacts of the design or implementation.

6.1 Tagged Pointers

Tagged-pointer based schemes cannot guarantee the integrity of pointer tags on programs that exploit low-level pointer representations. These programs may use the upper bits on the pointer for custom data storage, conflicting with the pointer tag usage of defense schemes. In-Fat Pointer is not designed to support these programs, and they may not execute correctly when instrumented by IFP.

Another problem is that tagged pointers on their own cannot ensure the integrity of pointer tags, unless either a pointer integrity scheme is deployed, or the full memory safety and type safety is enforced. While In-Fat Pointer stops spatial memory errors from corrupting pointers and their tags, temporal memory errors (i.e., use-after-free) and type confusion can still corrupt application data, including the data pointers. The future design of In-Fat Pointer may explore more optimized pointer tag bits allocation to embed a MAC on the tag to detect corruption on pointer values and tags.

6.2 Object Metadata Schemes

While the three object metadata schemes are designed to be complementary, their current design does not ensure full coverage for all possible objects that a program can allocate, and some design elements impose limits on their coverage or efficiency. The global table scheme has a capacity limit on the number of supported objects, therefore programs using too many global arrays can have some arrays not protected from overflow. The runtime initialization of MAC on local offset scheme metadata prevents their use on read-only global constant data. The design space for the subheap scheme is not fully explored. For small allocations, it is challenging for a compiler alone to identify whether a dynamic allocation is frequent enough or not so that the benefit of the subheap scheme’s metadata sharing can show up, and the evaluation shows that its current design has much larger overhead than the local offset scheme. For large allocations, the design would reserve so much virtual memory space that the system virtual memory setting needs to be adjusted for it. The future design of In-Fat Pointer should redesign
or propose workarounds for these problems.

### 6.3 Subobject Protection and Layout Tables

The type information in weakly-typed languages like C/C++ is not guaranteed to be reliable or available, and so does the subobject bound narrowing of In-Fat Pointer that depends on the availability and accuracy of the allocated type. The compiler optimizations can cause the type information to be inaccurate or missing. Programs using wrapper functions for dynamic allocation can prevent the compiler from knowing the use types of allocated objects when no whole-program or link-time optimization techniques are used. Besides, the allocated type may not be statically solvable when the allocators are called using function pointers. Comparing with traditional fat pointer schemes that only rely on the actual use type of the object for bound narrowing, In-Fat Pointer does not always guarantee subobject-granularity protection; in the worst case, the subobject-granularity protection is only guaranteed after a bound narrowing is made.

The current design of the layout table does not support arbitrarily complex types. First of all, because C/C++ programs can arbitrarily nest arrays of structs, the depth of the tree embedded in layout tables with bound narrowing paths are unbounded. However, a hardware implementation will likely impose a limit on the tree depth (and therefore the level of array-of-struct nesting in source C/C++ code) so that the layout table elements can be fetched only once into the execution unit, therefore programs with more nesting levels will not have full subobject-granularity protection. Secondly, the size of the layout table is constrained by the width of the subobject index on the pointer, therefore types with too many subobjects cannot be supported\(^1\). Future implementation of In-Fat Pointer may use aggressive program analysis techniques to emit layout table elements that are only actually needed, instead of conservatively emit all subobjects under the type. A study on application behavior for statistics on array-of-struct nesting and the number of subobjects will also be helpful for tuning design parameters (e.g., bit assignments) and guiding future design iteration of layout tables.

Because of the sequential fetching of layout table elements and bound narrowing operation replaying necessary when handling array-of-struct nesting, the current layout table design also results in a large variation in the latency for subobject bound narrowing. A future implementation may use better program analysis techniques to opportunistically expand applicable arrays as structs to reduce the depth of array-of-struct nesting.

---

\(^{1}\)The current implementation can support layout tables with at most 256 elements for the subheap scheme and 64 elements for the local offset scheme.
Chapter 7

Conclusion

It remains an unsolved problem for spatial memory safety defenses to achieve low overhead, high compatibility, and fine-grained protection granularity at the same time. This thesis presents In-Fat Pointer, a hardware-based tagged-pointer scheme as one more step toward solving the problem. Existing tagged-pointer schemes using object metadata can achieve both low overhead and high compatibility with hardware assistance, but they cannot enforce fine-grained pointer bounds. These schemes devote all tag bits for object metadata lookup, leaving no space for per-pointer metadata that is necessary for subobject bound enforcement. In-Fat Pointer improved the protection granularity to subobject bound by first using three complementary object metadata schemes to spare precious pointer tag bits, and then propose the layout table as the mechanism to utilize the spared pointer tag bits for subobject bound narrowing after the object bound is retrieved. The evaluation shows that In-Fat Pointer is effective in enforcing spatial memory safety, and it imposes performance overhead from 9% to 23% and memory overhead of less than 18%. Later works can expand the idea of complementary metadata schemes for more efficient metadata scheme specialization and re-purpose the saved pointer tag bits for other security policies.

7.1 Future works

The current design and implementation of In-Fat Pointer still need improvement to tackle the limitations described in Section 6.

**Improvement on the layout table.** The current design of the layout table only support C/C++ structs with a limited number of elements. The current implementation also cannot handle deep nests of arrays of structs. One of the future work is to move the compiler instrumentation to the link-time optimization phase so that all unused layout table elements can be optimized away and only the necessary elements are put into the table. In addition, the compiler may identify all pointers to deeply nested subobjects, and the compiler may perform aggressive program transform (including using inline fat pointers) around these pointers to avoid burdening the layout table when it can prove that the transform does not break compatibility with legacy code.

**Improvement on metadata schemes.** The implementation of current object metadata schemes exposed some unexpected performance impact and usage limitations. The subheap scheme implementation
is significantly slower than using the local offset scheme with the global table scheme for dynamic allocations. The current design does not provide an efficient scheme for protecting overflow (overread) on constant global data. These issues will be addressed by a better exploration of the possible design space of metadata schemes.

**Language support.** Current implementation of In-Fat Pointer only support C programs. However, C++ involves language constructs that are hard to efficiently support in the current design, for example, pointer to members, and pointer down-casting. In-Fat Pointer implementation will likely be extended to support C++ programs in future iterations.


