MIFP: Selective Fat-Pointer Bounds Compression for Accurate Bounds Checking

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

Eric Liu
University of Toronto
Toronto, Canada
ec.liu@mail.utoronto.ca

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

David Lie
University of Toronto
Toronto, Canada
david.lie@utoronto.ca

ABSTRACT

Bounds compression for fat pointers can reduce the memory and performance overhead of maintaining pointer bounds and is necessary for efficient hardware implementation. However, compression can introduce inaccuracy to the bounds, making certain out-of-bounds accesses undetectable. Although the security threat can be mitigated by padding the objects, no known mitigations can detect these out-of-bounds accesses deterministically.

We present MIFP, a method that automatically mixes both compressed and uncompressed bounds to preserve the performance benefits of bounds compression while ensuring accurate bounds checking. Given a program using a single fat pointer representation (e.g., all compressed bounds), MIFP performs whole-program analysis to expand potentially unsafe and inaccurate fat pointers such that they carry accurate uncompressed bounds. To minimize the number of pointers to expand, MIFP adds instrumentation on a per-allocation-site granularity; objects of the same type but different code allocation locations can have their pointer members transformed differently depending on how the pointers are used. We describe our algorithm and supporting data structures, and show that utilizing multiple fat-pointer representations reduces the runtime and memory overheads of uncompressed bounds by 79% and 93% respectively.

CCS CONCEPTS

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

KEYWORDS

Memory Safety, Fat Pointers, Bounds Checking, Program Transform

ACM Reference Format:
Shengjie Xu, Eric Liu, Wei Huang, and David Lie. 2023. MIFP: Selective Fat-Pointer Bounds Compression for Accurate Bounds Checking. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3607199.3607212

1 INTRODUCTION

Memory corruption in C and C++ programs is still a threat to the security of software systems. While these memory-unsafe languages provide features like pointers and explicit memory management that are convenient for high-performance or low-level applications, misusing these features can lead to arbitrary memory read or write vulnerabilities for malicious attackers. Due to the large existing C and C++ codebase, it is still desirable to develop defenses and mitigations against memory corruptions from the compiler, system, and hardware perspective, instead of relying on programmer manual effort [26].

Among the existing defense approaches, fat pointers are generally accepted as an effective defense against memory corruption from bad pointer arithmetic (referred to as spatial memory errors). To distinguish out-of-bounds accesses from valid accesses, fat-pointer schemes augment each pointer with a pair of addresses that describes the accessible range for the pointer (thus "fat" pointers). The scheme can then check each memory access using the pointer against the address range (known as *bounds*) to stop memory corruption. However, because pointer bounds are address pairs, a straightforward implementation of fat pointers will, at the very least, triple the size of pointers, thus increasing cache pressure and memory overhead.

To reduce the overhead from the pointer bounds, recent works adopt *bounds compression*, which exploits the redundancy among the upper address bits of the pointer and the bounds [13, 28]. Prior work has observed a 10% runtime reduction and a 30% DRAM traffic reduction on average from large programs after deploying bounds compression [28]. However, when the redundancy in high address bits is insufficient to represent the full bounds accurately, the bounds compression scheme will either fail [13] or will over-approximate the bounds [28], creating opportunities for undetectable memory corruption. Fundamentally speaking, because the compressed bounds must have a fixed size for efficient retrieval, the bounds compression scheme must be lossy in certain conditions regardless of the encoding. The accuracy loss will become worse when there are fewer bits available for compressed bounds compared with object size, for example (1) when the system is 32-bit instead of 64-bit, or (2) the scheme squeezes additional metadata along with the compressed bounds into a single word [28].

While prior work proposes mitigation techniques for the bounds accuracy loss, none of them can address all the security and performance concerns. Taking CHERI as the example, although the allocation bound overflow can be mitigated by padding the allocations so that only unused memory gets accessed, CHERI provides no mitigation for *sub-allocation* bound overflow [11]. This means it is possible to corrupt adjacent struct fields from an array inside the struct if the array is large enough. To address sub-allocation overflows, the scheme needs to modify the object memory layout to insert padding around array members, which can break compatibility with external code. Modifying the allocator and padding memory can also incur non-negligible memory overhead [9]. In addition, these mitigation techniques cannot stop access to padded memory. If an application contains an access control module (either for software resources like devices or for physical assets) and the permission data is stored in an array, interpreting the content in the padded memory as valid permissions can result in a compromise. It is necessary to preserve the bounds checking accuracy to address all the security concerns above.

In this paper, we present MIFP (Mixed Inline Fat Pointer), the first solution that ensures accurate bounds checking for bounds compression schemes by selectively using uncompressed bounds. We use whole-program static analysis to identify pointers for which the current pointer bounds are not sufficient to accurately prevent an out-of-bounds access, and then transform the program so that these pointers carry uncompressed bounds for accurate bounds checking[1]. MIFP is the first work that supports multiple fat pointer representations simultaneously in a hardware-assisted fat pointer scheme and *selectively* modifies pointer representation using compiler instrumentation instead of changing the programming language rules.

The design of MIFP solves three key challenges: (1) functionality: the algorithm should transform any fat pointer whose bounds inaccuracy can weaken security; (2) correctness: the algorithm must not introduce inconsistent object memory layouts in the presence of pointers to aggregate types or other pointers, and (3) granularity: the algorithm should not expand fat pointers that do not need expansion for security. To solve the functionality challenge, we use a value-flow-based abstract interpretation algorithm that uses solvable pointer properties (e.g., maximum bounds length) to find unsafe pointers whose compressed bounds are inaccurate, then identifies pointers along the dataflow to propagate uncompressed bounds. To satisfy the correctness and granularity requirements, we propose an *extended points-to graph* to support the analysis. The graph integrates the points-to information with dataflow and solves type constraints from points-to relationships during graph construction, therefore it can map pointers found by the abstract interpretation to all pointers and objects affected by them. This property simplifies using the abstract interpretation results for instrumentation. When an aggregate type (e.g., struct) contains pointers, prior work transforms all instances of the same aggregate type identically [19]. In contrast, our extended points-to graph achieves per-allocation-site granularity for transforming object types. To our best knowledge, MIFP is also the first work that achieves per-allocation-site granularity for such a type transform.

[1]MIFP requires hardware modifications if the base scheme does not support uncompressed bounds.

We built our prototype implementation for CHERI [28] on the 64-bit RISCV platform and evaluated the prototype with 14 programs from Olden, PtrDist, and CoreMark. The prototype implementation incurs 8% run time increase and 2% memory usage increase from the CHERI baseline in which the programs only use compressed bounds. Compared with the 35% run time increase and 28% memory usage increase for using uncompressed bounds only, by selectively expanding only pointers that require the additional accuracy, and enabling the mixed use of fat pointer widths within a program, MIFP achieves the same security level as disabling bounds compression but with only 21% of the performance overhead and 7% of the memory overhead.

The contributions of this paper are:

- We describe MIFP's whole-program analysis and transform algorithm for utilizing multiple fat pointer representations without programmer effort.
- We show the design of MIFP's extended points-to graph, a data structure that assists our algorithm, and we explain how it helps solve challenges in our transform that are hard using traditional dataflow analysis.
- We implemented MIFP based on CHERI on 64-bit RISCV and we present an evaluation of MIFP with selected benchmarks from Olden, Ptrdist, and CoreMark running on FPGA implementation.

The remainder of this paper is organized as follows. First, Section 2 provides the background and related work. Next, Section 3 presents the design of MIFP, including the analysis algorithm and the extended points-to graph. Section 4 describes our implementation based on CHERI on 64-bit RISCV and Section 5 evaluates the correctness and performance of our implementation. Finally, Section 6 discusses limitations and strengths of MIFP and Section 7 concludes this paper.

2 BACKGROUND AND RELATED WORK

In this section, we introduce related work and the background. We first describe fat pointers and bounds compression in Section 2.1, then introduce CHERI [28] in Section 2.2, which is an example fat-pointer scheme that the MIFP prototype builds upon. Next, we compare MIFP with related work on pointer-based memory safety defenses and program transforms in Section 2.3. Finally, we introduce two graph data structures that MIFP builds on in Sections 2.4 and 2.5. We use the code example in Figure 3a throughout this paper for consistency.

2.1 Fat Pointers and Bounds Compression

Fat pointer is a well-understood defense against spatial memory errors. As shown in Figure 1, fat-pointer schemes associate a pair of addresses `<base, top>` with each pointer (originally only an `address` value) that constrains the valid access range for the pointer, and they can catch any dereference outside the range with bounds checks. The bounds can either be stored next to the pointer [5, 28, 29] or in distinct memory regions [6, 16, 17, 20]. While good for security,

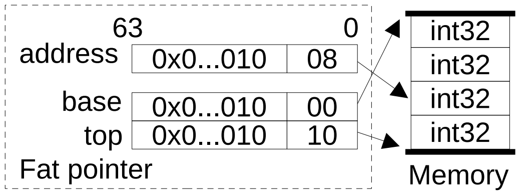


Figure 1: Example fat pointer pointing to a 4-element `int` array at 0x1000.

pointer bounds impose additional memory and performance cost for accessing and checking; efficiently performing these operations can reduce the overhead of fat pointer schemes.

Bounds compression is based on the observation that the upper bits of the `address`, `base`, and `top` have redundancy available for compression. In the example in Figure 1, for an array with four 32-bit integers, only the lowest 5 bits of the address and the bounds differ, and all the upper bits are redundant. Therefore, only the lower bits of `base` and `top` are necessary to encode the bounds; the upper bits can come from the `address`. Bounds compression schemes typically use a fixed-width floating-point style encoding, with the "significand" part representing the lower bits of `base` and `top`, and the "exponent" representing the start position of the encoded lower bits. This compression strategy usually works well because most of the allocated objects are small [3, 34], so the number of differing lower bits among `address`, `base`, and `top` are limited.

2.2 CHERI

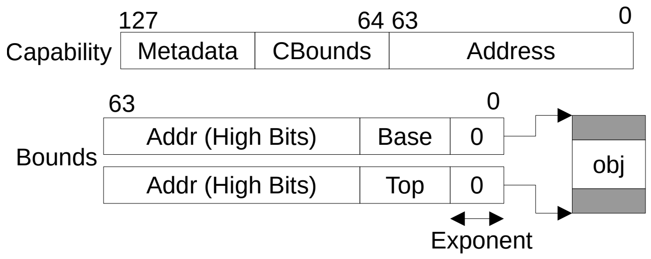


Figure 2: Simplified view of CHERI Compressed Capability Encoding.

CHERI [28] is a hardware-assisted fat pointer scheme with bounds compression. Figure 2 shows a simplified view of CHERI's fat pointer (referred to as capability pointers). These fat pointers can reside in registers, and user-level instructions are available to manipulate them. Bounds checking is part of the semantics of load and store instructions in CHERI, so normal code does not need explicit bounds checking instructions. The compressed bounds (`CBounds` in the figure) encode (1) the lower bits of `base` and `top` for the start and end address of the permissible range, and (2) an internal exponent representing the starting position of the bounds bits. CHERI uses dedicated instructions that update the compressed bounds during pointer arithmetic. If a pointer arithmetic operation pushes the address so far away from the original bounds that the original bounds cannot be encoded, it invalidates the capability. When the bounds length (`top - base`) is small and the pointer is in bounds, the internal exponent will be zero and the bounds are accurate. However, when the bounds length exceeds a representable limit, the compressed bounds cannot encode all distinct lower bits of base and top, so the internal exponent will be non-zero and the lower bits of the computed base and top will be zero-filled. In this case, if the object's start or end address is not sufficiently aligned, the bounds will cover the extra memory before or after the object (the grey zone in Figure 2), where out-of-bounds accesses can evade checks. If these memory ranges happen to store critical data, the attacker can read or corrupt them silently. On a 64-bit system, a 128-bit capability can start to enlarge bounds when the object size exceeds 4096 bytes; an array of 4097 bytes will leave 7 bytes over the upper bound unprotected [11]. The CHERI paper suggests padding the allocations in the dynamic allocator so that out-of-bounds accesses to the padded memory are benign.

Besides spatial memory safety, CHERI also implements other security policies. (1) Pointer Integrity with tagged memory: all capability registers and each capability-aligned word in memory has one tag bit indicating whether the location contains a capability and the instructions can check or manipulate the tag bit to detect corruption of capabilities. (2) Capability model: CHERI includes additional metadata (represented as `Metadata` field) on pointers to enforce fine-grained memory access control policies, including software compartmentalization. However, while CHERI can enforce additional security policies with the additional metadata bits, it also reduces the available bits for compressed bounds, which makes bounds accuracy loss worse.

2.3 Related Work

Storing bounds inline with the pointer like CHERI does has a performance advantage compared with other approaches because of the spatial locality of the pointer and bounds. With bounds compression, CHERI can use a single memory access to load the entire fat pointer, achieving the minimal load-to-use delay. However, because pointer bounds occupy extra space and object memory layouts may need modification, fat-pointer-enabled code can be incompatible with legacy binaries. It is also challenging to selectively transform pointers without breaking the program. Shakti-MS [5] transforms all pointers in the program into fat pointers. CHERI [28] either compiles all pointers to fat pointers, or only pointers with programmer annotation[2]. CCured [19] uses a type inference algorithm in the compiler frontend to assign different types to pointers on a per-pointer basis so `SAFE` pointers are left unmodified while other types carry extra metadata. However, if a pointer is declared inside a struct type, the type inference algorithm of CCured does not distinguish between the pointer values in different struct instances, so all parent objects with the same source type will also have the same result type. This also means that if one pointer member in a struct is considered unsafe and now carries metadata, all struct instances will reserve space for the metadata, even if the pointer is never accessed in other instances. Compared with CCured, MIFP can transform fewer unnecessary pointers; our algorithm can transform objects with the same source type differently depending on how they are used.

[2]Corresponding to CHERI's pure-capability mode and hybrid mode, respectively.

Besides transforming pointer layouts, recent work on compiler optimizations also pursues more efficient bounds checking. A program instrumented with WPBound [31] skips the bounds checking in a loop if it passes an instrumented stronger check outside the loop. PICO [12] consolidates bounds checking so that fewer checks are needed and places them at less frequently executed locations.

Another popular approach for maintaining bounds is to store them in disjoint memory locations. This reduces the compatibility problem with legacy code because object memory layouts do not need modification, but at the cost of extra overhead from locating bounds for each pointer. Intel MPX [20] uses a two-level directory structure to map the address of each pointer to the address of its bounds, and after loading a pointer, the program needs two extra non-contiguous loads to retrieve the bounds. WatchdogLite [16] uses a linear mapping from pointer to metadata address, so it only uses one extra load for the bounds at the cost of wasted virtual address space.

One extreme of bounds compression is to perform aggressive lossy compression such that the bounds fit into the unused high address bits on a pointer. These on-pointer metadata bits are referred to as pointer tags. Low-Fat Pointer [14] carefully places objects and adds padding so that the pointer can be checked using the on-pointer metadata from the high address bits. Later works like FRAMER [18], In-Fat Pointer [30], and HeapCheck [22] use the pointer tag to locate another in-memory metadata for more accurate checking. Another use of pointer tags is to implement probabilistic memory safety checks with memory tagging [24]. ARM MTE [2] associates every 16-byte memory with a 4-bit tag, and each pointer can carry a 4-bit tag in the high bits. During a dynamic allocation, the allocator assigns the same 4-bit value to the pointer and the allocated memory. Therefore, the hardware can detect out-of-bounds access by checking whether the pointer tag and the memory tag match.

Existing work has explored extending memory padding to mitigate sub-allocation bound overflow. Califorms [23] inserts padding between struct fields to detect such overflows. No-Fat [9] uses structure splitting [8, 21, 32] to promote arrays in structs into separate allocations so that the sub-allocation bound overflow can be mitigated similarly by padding these array allocations. No-Fat reports a 6.66% memory overhead for the allocator change [9].

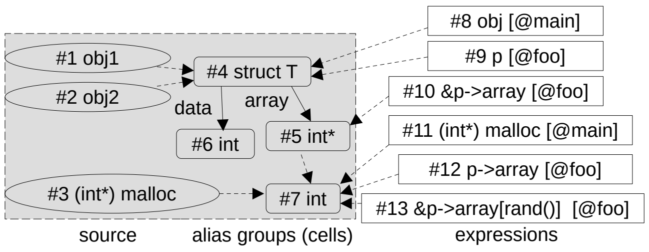
2.4 Points-to Graph

A points-to analysis can produce a points-to graph to describe the relationship between pointers and objects in a program. MIFP adapts the recent cell-based field-sensitive (CFS) points-to analysis [27] for analysis and transform. Figure 3 shows an example CFS points-to graph. We highlight the essential nodes and edges representing memory states with the grey box. Nodes #8-13 help show the relationship between the points-to graph and subsequent graphs including the SVFG (Section 2.5) and MIFP's extended version (Section 3.2).

In Figure 3b, nodes without incoming edges represent pointer-typed program expressions, including nodes #1-3 representing object allocations (referred to as *source* nodes) and nodes #8-13 representing pointer values in the program. The rest of the nodes (#4-7) represent *alias groups*, which are sets of indistinguishable memory

```
 1 struct T { int* array; int data; };
 2
 3 int foo(struct T* p) {
 4   return p->array[rand()];
 5 }
 6
 7 int main(int argc, char** argv) {
 8   struct T obj1, obj2;
 9   struct T* obj = rand()%2 ? &obj1 : &obj2;
10   size_t size = sizeof(int)*argc*1000;
11   obj->array = (int*) malloc(size);
12   memset((void*)obj->array, 0, size);
13   return foo(obj);
14 }
```

(a) Code Example



(b) Graph Representation

Figure 3: CFS Points-to Graph

locations (or *abstract objects*). In addition, alias group nodes in CFS analysis are typed, and they are named *cells*. Each node can have (1) `points-to` edges from pointer to pointee (shown as dashed arrows) and (2) `contains` edges from aggregate type objects (e.g., structs and arrays) to the member fields (shown as solid arrows). For example, node #4 represents the `struct T` for both `obj1` and `obj2` allocated at line 8, so there are `points-to` edges from #1 and #2 to node #4. The shared `struct T` contains the `array` and `data` member, therefore the two members are represented by distinct nodes #5 and #6 with `contains` edges from node #4. Nodes #8-13 outside the grey box represent scalar pointer values, so they only have `points-to` edges to cell nodes.

One important property of a CFS points-to graph is that each node has at most one outgoing `points-to` edge. If a pointer can point to multiple cells, those cell nodes will be merged. For example, since `obj` at line 9 can either point to the `obj1` or `obj2`, the graph construction algorithm will merge the cell node for `struct T` from `obj1` and the node for `obj2`. Therefore, there is a single node #4 representing the `struct T` for both of them.

2.5 Sparse Value-flow Graph

A sparse value-flow graph (SVFG) for pointers describes the interprocedural value flow (similar to SSA's def-use chain) of all pointers. It combines the value flow directly available from the program source (i.e., def-use chain in LLVM IR) and from analysis results that look through function calls and memory load/stores. The MIFP

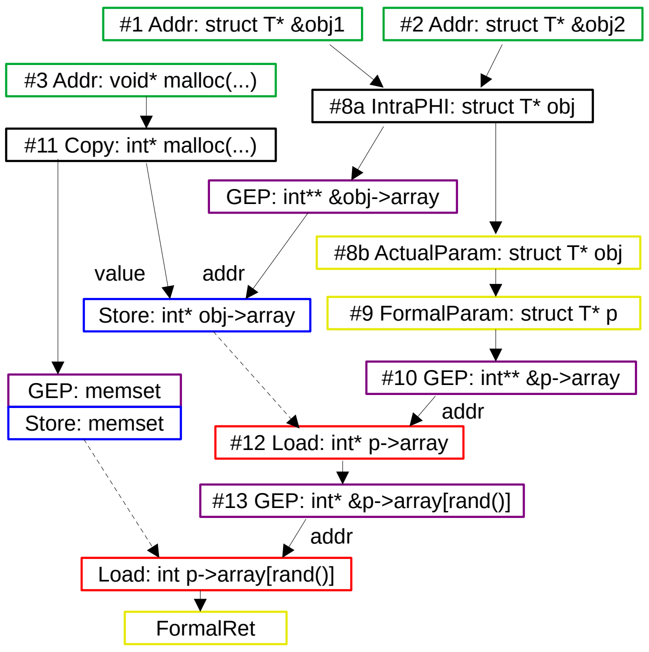

#1 Addr: struct T* &obj1
#2 Addr: struct T* &obj2
#3 Addr: void* malloc(...)
#8a IntraPHI: struct T* obj
#11 Copy: int* malloc(...)
GEP: int** &obj->array
#8b ActualParam: struct T* obj
value
addr
Store: int* obj->array
#9 FormalParam: struct T* p
#10 GEP: int** &p->array
GEP: memset
Store: memset
addr
#12 Load: int* p->array
#13 GEP: int* &p->array[rand()]
addr
Load: int p->array[rand()]
FormalRet

Figure 4: Sparse Value Flow Graph (SVFG)

implementation uses the SVF [25] library to perform points-to analysis and produce an SVFG for subsequent analysis. Figure 4 shows a simplified SVFG created from the code in Figure 3a. Each node represents a value, a memory location, etc. at one program location, and each edge represents a value flow. If an SVFG node corresponds to a pointer value in Figure 3, we use the same node index for the SVFG node. If an SVFG node does not map to any node in Figure 3, we leave it unlabeled.

For each memory allocation in the program, SVF creates an `addr` node (plotted in green boxes) to represent a pointer to the allocated memory. For example, nodes #1 and #2 represents the address of `obj1` and `obj2`, and node #3 points to the dynamic allocation of `int` at line 11. The `copy` node #11 represents the pointer cast on the return value of `malloc()`. The `IntraPHI` node #8a represents the result `obj` being selected from either `&obj1` or `&obj2`. Pointer arithmetic or struct member address computation is represented by `GEP` nodes, shown as purple boxes. For each load and store, SVF also creates `load` and `store` nodes (red and blue boxes respectively) representing the value being loaded or stored. Therefore, the single statement at line 4 is decomposed into (1) compute the address of the `array` member from `p` (node #10), (2) load the `array` pointer (#12), (3) compute the address of the indexed array element (#13), (4) load the array element, and (5) return it. SVFG uses *direct* edges (plotted in solid black lines) in the graph to represent direct value flows in the source program. Each `store` node can have two incoming direct edges, one for the address of the store, and the other for the value being stored. For example, the unlabeled `store` node to `obj->array` takes the value from node #11 and the address from the unlabeled `GEP` node `&obj->array` below node #8a. Each `load` node only has one incoming direct edge representing the address. Besides direct edges, SVFG contains *indirect* edges representing value flows through memory or indirect function calls. SVF plots indirect edges in dashed lines. For example, the path from the unlabeled store to `obj->array` to load #12 means that the stored value can reach that load. These incoming indirect edges to load and store nodes represent all possible values that can be read by the load or overwritten by the store. Yellow boxes represent inter-procedural values like function parameters (`ActualParam`/`FormalParam`), return values (`FormalRet`), and memory locations[3]. `FormalXXX` represent callee-side values (e.g., formal parameter `p` at node #9) and `ActualXXX` represent caller-side values (like passed parameter `obj` at node #8b).

2.6 Value Flow Analysis

MIFP uses multiple (sparse) value flow analyses on SVFG to solve certain properties on pointers using abstract interpretation. This is similar to traditional abstract interpretation or dataflow analysis except the algorithm runs on a value flow graph (e.g., SVFG or LLVM def-use chains) and handles one value at a time, instead of on a control flow graph where we handle all values in the basic block in each iteration. Assuming we perform value flow analysis on the SVFG shown in Figure 4, we will associate a variable x with each pointer-type SVFG node. The analysis involves (1) initializing the variables x for each node, (2) selecting analysis-dependent starting nodes and adding them to a worklist, and (3) running the main loop until the worklist is empty. In the main loop, the algorithm (1) pops one node from the worklist, (2) updates the variable of other nodes with value flow edges to the popped node, and (3) if any node has the variable changed, adds them to the worklist. We use two minimal examples based on the code in Figure 3a to explain the algorithm.

Forward analysis. Assume we want to derive a statically safe constant range for each pointer so that later on we can check if a pointer dereference is safe by comparing the accessed range with this statically safe range. For example, we want to prove that loading `p->array` at line 4 in Figure 3a (node #12) is always safe because it must be within a valid `struct T` object. For simplicity, we focus on the value flow path from #1, #2, #8a, #8b all the way to #12. The variable x for each pointer will be either a constant interval $[a, b)$ representing the safe range of bytes or a special unknown value. If a pointer is initialized by external code, we conservatively assign $[0, 0)$ as the safe range. Initially, all pointers have an unknown safe range except node #1 and #2 which have $[0, 12)$, assuming 8-byte pointers, 4-byte int values, and `struct T` is 12 bytes in size. The worklist is initialized with $\{\#1, \#2\}$. Assuming a FIFO worklist, in the first iteration, node #1 is popped and node #8a copied the safe range $[0, 12)$. Node #8a is also added to the worklist because its safe range is updated. In the second iteration, node #2 is popped and the algorithm will compare its safe range with Node #8a. Because their safe ranges are identical, no change is made. In the following iterations, the safe range $[0, 12)$ will be copied to node #8b, #9, #10 in this order. We can show that load #12 is safe because the accessed range $[0, 8)$ is within the range $[0, 12)$.

Backward analysis. Assume we conclude that loading the `int` from `p->array` at a random index is not statically safe so the address

[3]Nodes not essential in the example are omitted from the figure, including `ActualIn`/`FormalIn`, `ActualOut`/`FormalOut`, and `ActualRet` nodes.

pointer represented by node #13 needs bounds. Suppose we want to insert accurate checks for pointer dereferences which are not statically safe, and we use a naive backward analysis to find which pointers should be expanded to carry bounds. We focus on the path from node #3, #11 to #12 and #13. The variable x is an enum `{raw, fat}` where `raw` means the pointer has no bounds and `fat` means the pointer carries bounds. Initially, all pointers are `raw` except #13 because it is used in the potentially unsafe access, and the worklist is initialized with `{#13}`. In each iteration of the loop, for a `fat` pointer node, we change all source pointer values to `fat` as well. Therefore, node #13 will cause #12 to be fat in the first iteration, and pointers along the way till node #11 and #3 will be `fat` in subsequent iterations. Other nodes outside the path are not affected because they do not represent the same pointer value. For example, node #10 will not be `fat` although node #12 is `fat` because the load itself is safe so the address requires no checking; only the loaded pointer should be expanded with bounds. Note that the described algorithm is incorrect on its own because it cannot identify all pointers in memory that must be expanded; Section 3.2 provides the fix.

3 DESIGN OF MIFP

Given any fat-pointer defense with compressed bounds support, MIFP's goal is to maintain the performance benefit of bounds compression without losing accurate bounds checking. MIFP achieves this by identifying all pointers whose bounds may be inaccurate and miss out-of-bounds accesses, and transforming these pointers to have uncompressed, fully accurate bounds. If an expanded pointer is address-taken, embedded in other objects, or passed as function arguments or return values, MIFP also modifies all affected code to accommodate the extra bounds. Section 3.1 gives the overview of the algorithm.

MIFP stores uncompressed bounds inline with the compressed fat pointer for efficiency. However, it is difficult for compiler transforms to modify memory layouts of address-taken objects because the transform would need to find and update all code accessing the object using the original layout. To solve this challenge, MIFP uses the extended points-to graph to track the object memory layouts in the compiler IR and guide the subsequent transforms. Section 3.2 describes the extended points-to graph in depth.

There are two components in MIFP's implementation: (1) whole-program analysis and transform to reserve space and insert instructions for the uncompressed bounds, and (2) support for assembling and executing instructions related to uncompressed bounds in the fat-pointer defense (if not already present), including modifying (a) the compiler backend and (b) the execution environment like QEMU or hardware. Since uncompressed bounds support is specific to the fat-pointer defense, in this section, we focus on the analysis and transform algorithm. Details of the uncompressed bounds support in our prototype implementation for CHERI on 64-bit RISCV are described in Section 4.

Limitations. In the extended points-to graph, our MIFP prototype uses LLVM IR types on cell nodes to represent the memory layout of objects. Unions and arbitrary pointer casts are not fully supported because LLVM IR does not contain union types and our prototype has limited support for conflicting types. Section 6 lists additional design or implementation limitations that may impact certain applications.

Terminology. In the rest of this paper, a pointer's *width type* denotes the bounds representation of this pointer (i.e., having compressed bounds, uncompressed bounds, or no bounds). A raw pointer has no bounds. We say a compressed fat pointer is *wider* than a raw pointer, and an uncompressed fat pointer is wider than the other two. *Expanding* a pointer means changing the width type of the pointer to a wider and thus more accurate one (likely to an uncompressed fat pointer). A pointer's *type* includes both the width type and the element type (i.e., the type of data it points to). Because the types of pointers and objects determine their memory layouts, we transform the types of these program constructs when expanding pointers.

3.1 Overview of MIFP Whole-program Transform

Algorithm 1 MIFP whole-program transform

```
$M \leftarrow ReadIR()$                                  {Read the input IR}
$G \leftarrow BuildGraph(M)$                  {Build extended points-to graph}
$P_u \leftarrow FindUnsafePtr(M, G)$                             {Step 1}
$D : V_{ptr}(G) \mapsto W \leftarrow \emptyset$                  {Pointers to expand}
$S : V_{ptr}(G) \mapsto W \leftarrow BoundsAccuracy(G, M, D)$    {Step 2}
repeat
    $D \leftarrow FindExpandingPtr(G, M, S \cap P_u, D)$         {Step 3}
    $S \leftarrow BoundsAccuracy(G, M, D)$                       {Rerun step 2}
until $S \cap P_u \subseteq D$                                   {Loop until converge}
$T \leftarrow EmitTypes(G, M, D)$                                {Step 4}
$M' \leftarrow Instrument(M, G, T)$                              {Step 5}
Write $M'$                                               {Output transformed IR}
```

MIFP introduces a new whole-program transform on the compiler intermediate representation (IR) to make space for uncompressed bounds and to insert manipulation and checking instructions in user programs. Performing the transform in the IR stage enables MIFP to reuse existing compiler analysis passes to decide which pointers to expand. First, after all the source files are compiled into IR, we collect all of them and create a whole-program IR. Then, MIFP algorithm reads the whole-program IR and applies the whole-program transform to it. The output IR can then be passed to later compilation pipeline stages.

Algorithm 1 shows the overall procedure of the transform. W represents the set of all width types (raw pointer, compressed fat pointer, and uncompressed fat pointer), and $V_{ptr}(G)$ represents all pointer-type nodes in the graph. Before the whole-program transform starts, we first build a new graph structure called *extended points-to graph*, and then run the rest of the transform that depends on the graph. To better explain the problems this graph solves, in the rest of this subsection, we will first discuss the goal and methodology of each stage in the algorithm and then formally introduce the graph in Section 3.2. Because the extended points-to graph contains nodes representing pointers and objects and can map any pointer in the program to one of the nodes, the discussion below will use "a pointer" to refer to "a pointer-typed node in the extended points-to graph". Note that while the pseudocode in

Algorithm 1 shows all intermediate states as distinct maps and sets, we can implement these states as node attributes in the extended points-to graph.

Step 1: find unsafe pointers. The first step of the algorithm in Algorithm 1 needs to identify pointers used in dereferences that cannot be proven statically safe. These pointers are assumed unsafe and are put into the set P_u for later checking. We use the approach from earlier work [1]: first solving a safe range for each pointer from object allocations, then checking each dereference against the range. This is similar to the forward analysis example described in Section 2.6.

Step 2: solve bounds accuracy. After finding all unsafe pointers, step 2 determines whether each pointer is wide enough so that bounds are always accurate for checking. In Algorithm 1, the result of step 2 is a partial mapping S that maps each pointer in it to the minimum accurate width type that ensures bounds accuracy. Only the pointers *not* wide enough in the input program will be present in S. Therefore, $S \cap P_u$ represents unsafe pointers that require expansion. Because we will find pointers for expansion (D) in step 3, which can in turn change the type size of elements/objects that contain such pointers and invalidate the results computed in this step, we run steps 2 and 3 iteratively until no more pointers require expansion ($S \cap P_u \subseteq D$). When initializing the maximum bounds length from allocation sizes, the code will consider the type change due to pointer expansion (available after step 3) starting from the second iteration.

Finding the minimum accurate width type is inherently specific to the bounds compression scheme. For our implementation on 64-bit RISCV with CHERI, we use a forward analysis to compute the fixed-point solution of (1) maximum bounds length b (non-decreasing, ∞ for unknown) and (2) minimum bounds alignment a (non-increasing, 1 for unknown) for unsafe pointers, and when the pointer uses compressed bounds, we assume the bounds are always accurate if $b \leq 4096$ or $b \leq a \times 2^{10}$. For example, because CHERI rounds up any bounds length between 4097 and 4104 bytes to 4104 bytes, the compressed bounds for a 4104-byte object are accurate because the bounds are 8-bytes aligned, but those for a 4103-byte object are *not* accurate because the bounds are only 1-byte aligned and the length will be approximated to 4104 bytes. We start from the allocation sites (which are also pointer sources), compute the initial bounds length and alignment for pointers from allocations, then use a forward analysis to solve the fixed point solution for the remaining pointers. When computing the initial bounds length for allocations, we walk the extended points-to graph to see if the allocated type has any member pointer marked for expansion (i.e., the pointer is in D in Algorithm 1), and compute the new type size when needed.

Step 3: find pointers for expansion. After we find unsafe pointers whose bounds are not accurate enough ($S \cap P_u$) we identify all pointers that require width type expansion. The result D is a partial map that contains all pointers that need expansion. To ensure the uncompressed bounds can propagate from the pointer source (i.e., allocation sites) to the use sites, we use a backward analysis that starts from the unsafe pointers at dereference site, traverses along the value flow path toward the pointer source (allocation sites), and marks all pointers along the way for expansion including ones without unsafe use. This is similar to the backward analysis example described in Section 2.6. If the pointer is loaded from memory, we expand the in-memory pointer and all pointers stored to the memory location. If the pointer is a function parameter, we expand all call site arguments to this parameter. This ensures that the accurate bounds can reach the use site for checking. As described above, steps 2 and 3 are run iteratively until we find no more pointers to expand.

Step 4: emit types. Once all pointers requiring expansion are found, we create the new types in the compiler IR for all program constructs (including objects and function prototypes) that need type changes. In Algorithm 1, the output is a map T from every node (including non-pointer type nodes) in the extended points-to graph to their new types. Because the algorithm heavily depends on the extended points-to graph, we will describe it in Section 3.2.2.

Step 5: instrumentation. After all the previous analysis steps, the last step is to apply all the transformations. This involves (1) replacing the type of global variables and function prototypes, (2) replacing the type of dynamic allocations and local variables, and finally (3) updating the users of these values to use the correct type. This step also involves defense-specific logic, like the representation and manipulation of uncompressed bounds in compiler IR, and inserting bounds checks that use uncompressed bounds.

3.2 Extended Points-to graph

There are two challenges in the algorithm that cannot be solved with value flow analyses alone. These challenges lead to our design of the extended points-to graph. We first describe the graph in Section 3.2.1, then we describe how the graph assists the analysis and transform in Section 3.2.2. Before further discussion, we first list the encountered challenges below to highlight the motivation for the graph.

Handling points-to constraints. When we transform the types of pointers and objects in step 5, the transformation must satisfy the following *points-to constraint*: each pointer should only point to objects with a consistent type. In other words, if one points-to object needs a type change, all the other points-to objects of the same pointer should be changed consistently. In the code example in Figure 3a, because the pointer `p->array` at line 4 can either be `obj1.array` or `obj2.array`, the algorithm should either expand both of `obj1.array` and `obj2.array` or expand none of them. Otherwise, the element type of `obj` and `p` becomes inconsistent because the `array` pointer inside the `struct T` instances have different width types. If the inconsistency is allowed, later instrumentation will introduce type errors and create incorrect programs because the memory layout of the points-to objects can be different. Although walking the value flow can help us find the pointers to expand, it alone cannot avoid all such inconsistencies. To solve this problem, we need the *points-to* information of `p` so that expanding either pointer can inform us to also expand the other.

Bridging pointer expansion and type change. Step 4 of the algorithm needs to determine the static type of all program constructs, including function prototypes and in-memory objects. Step 2 of the algorithm also needs to recompute the allocation sizes if there are pointers getting expanded in the allocation. These tasks require a new data structure that can translate pointer expansion to type changes, including the type change of the pointer and all other types depending on it. For example, if a struct type `S` contains a pointer that needs expansion, not only does the struct type `S` need to be transformed but also any pointers to the struct (`S*`) and any parent struct embedding the struct `S`. If a function takes a pointer to the struct `S`, the function prototype also needs to change. The new data structure must handle the chain reaction of type changes so that the algorithm can compute the correct sizes in step 2 and emit the correct types in step 4.

3.2.1 *Graph Definition.*

To tackle the challenges in the algorithm, MIFP designs the extended points-to graph to integrate value flows with points-to information, and handle type inter-dependencies for all program constructs like functions or in-memory objects. The graph is designed to work with another value flow graph, for example, the compiler IR in Static Single-Assignment (SSA) form or the Sparse Value-Flow Graph (SVFG). Figure 5a shows an example extended points-to graph from the code in Figure 3a. Essentially, the extended points-to graph extends a CFS points-to graph in Figure 3b with (1) additional expression nodes (not shown) and edges from value flows and (2) function-type cells as if they are structs with parameters and return values as members. The new nodes are highlighted in the grey region.

Nodes. First, we have one *source* node on the left side of the graph (node #1-3 and #14) for each variable declaration, object allocation, and function declaration. This is the same as existing points-to analyses that use allocation sites to name alias sets except for new source nodes from function declarations. Next, we have *expression* nodes on the right side (node #8 to #13) for pointer-type program expressions. Besides expressions analyzed by the points-to analysis, we also create nodes from the value flow graph (e.g., an instruction producing a pointer in compiler IR or an SVFG node). The remaining nodes in the middle (#4-7, #15-17) represent cells (or alias groups) for in-memory data and function prototypes; each cell node is a partition of all objects or functions in the program with the same type. All cell nodes and expression nodes are typed while the source nodes are not. The prototype implementation uses the LLVM IR types for all typed nodes. The "pointer-type nodes" in Section 3.1 refer to both cell nodes and expression nodes.

Edges. First, each source node has an edge to the cell representing the allocated type. This includes the edges from #1-2 to #4, from #3 to #7, and from #14 to #15 in Figure 5a. In the extended points-to graph, we treat function declarations as allocations as well, so function `foo()` has node #14 and its type in node #15. Second, similar to existing points-to graphs, there are `contains` edges from aggregate types (e.g., #4 for `struct T`) to members (#5-6 representing the `array` and `data` field), and `points-to` edges to cell nodes from expression nodes or other cell nodes. These edges are drawn with solid black lines and dashed black lines respectively. Because we treat function types as if they are aggregate types, we also create cell nodes #16 and #17 to represent the parameter `p` and the return value respectively and add `contains` edges from #15 to both of them. Lastly, the extended points-to graph includes value flow edges as well: store edges (blue) and load edges (red) connect the cell and the pointer-type expression nodes being stored or loaded, respectively. When there is a function call (e.g., at line 13), we add store edges from each argument (#8) to the cell for the parameter (#16), and load edges from the cell to the parameter expression (#9) in the function body. The reverse is true for returned pointers; we treat it as storing the formal return value to the cell and loading it from the cell inside the caller.

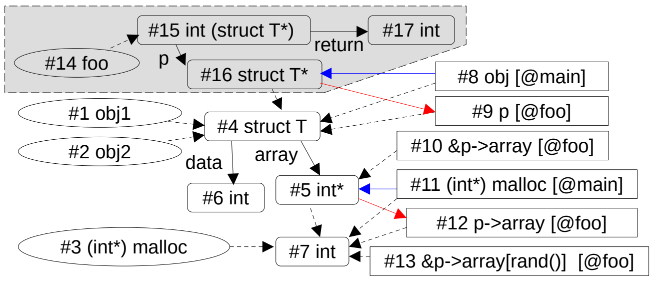
Properties. First, we inherit the property from the points-to graph that each pointer has at most one outgoing `points-to` edge. In the example, because `p` can point to either `obj1` or `obj2`, the two objects will share the same cell node #4 so that `p` points to this single cell node. This sharing also makes `&p->array` pointing to the single cell node #5. Expanding node #5 expands both `obj1.array` and `obj2.array`, so there will be no inconsistency between their types. Similar to existing points-to analyses, we merge cell nodes during graph building to enforce this constraint. Compared with the traditional constraint-solving approach where points-to constraints (e.g., the `TYPEEQ` constraint in CCured [19]) are solved roughly at the same time with other constraints, the extended points-to graph solves these constraints once during graph construction and no subsequent analysis (step 3 and 4 of the algorithm) needs to handle them again. After merging cell nodes for the points-to constraints, any remaining cells can have pointer expansion and type change results independently.

Second, the extended points-to graph captures all value flow edges from function calls and memory loads and stores. In other words, all value flows *not* directly available from the def-use chains in the compiler IR will be captured in the graph. This ensures that when we walk the value flow graph, whenever a pointer in memory or a function prototype is visited, we must have a corresponding node in the extended points-to graph. These common nodes connect the value flow with points-to information so that our algorithm can utilize both seamlessly.

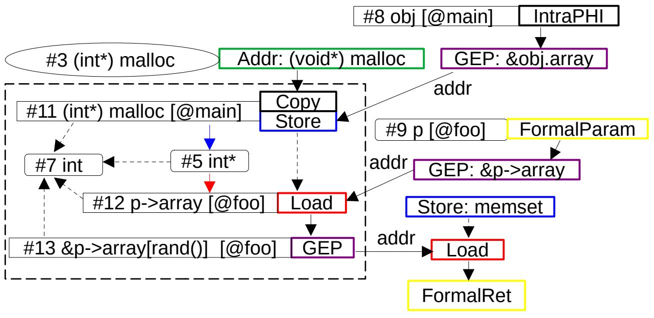
3.2.2 *Using the Extended Points-to Graph for Transform.*

Changes to value flow analysis. All value flow analyses in steps 1-3 from Algorithm 1 are modified to work on a composite graph that essentially merges the value flow graph (SVFG or LLVM def-use chain) with the extended points-to graph. All pointer-type nodes from the value flow graph can be added as expression nodes, and we maintain the mapping from the expression nodes to their backing value flow nodes. We discard non-pointer type nodes because we only analyze pointer value flows. The extended points-to graph does not define edges between two expression nodes; all such edges are from the value flow graph. However, if a value flow edge maps to a load or store edge in the extended points-to graph, the load or store edge in the extended points-to graph takes priority. When performing the value flow analysis for pointers, we associate variables to pointer-type cell nodes and expression nodes in the extended points-to graph. We query the value flow graph to get edges between two expression nodes and query the extended points-to graph to get edges between two cell nodes or edges between a cell node and an expression node.

Finding pointers to expand (step 3) revisited. Figure 5b shows part of the composite graph used by step 3 (Finding pointers to expand)



(a) Extended Points-to Graph



(b) Composite graph (selected view)

Figure 5: MIFP Extended Points-to Graph and its Composition with SVFG

in Algorithm 1. The backward analysis example in Section 2.6 is adapted to use the composite graph. We use the same code example from Figure 3a and plot the extended points-to graph nodes and SVFG nodes side-by-side if they are related. The expression nodes and cell nodes enclosed by the dashed box (except node #7 representing the point-to object) represent the pointers that will be labeled for expansion at the end of the backward analysis. When step 3 starts, because the load on `&p->array[rand[]]` is unsafe and the compressed bounds may not be accurate, node #13 will be marked as requiring expansion and enqueued to the worklist. In the first iteration, the algorithm walks the edge from Load to GEP, marks node #12 for expansion because node #13 will be expanded, and enqueues node #12. Because the graph creation algorithm maps the SVFG edges from #11 to #12 to edge #11 to #5 followed by edge #5 to #12, in the second iteration, the algorithm walks the load edge from #5 to #12 and updates #5. Then, in the third iteration, node #11 is updated from node #5. This allows the algorithm to identify that the pointer represented by node #5 requires expansion. If there are other pointers sharing the same cell #5 because of points-to constraints, the use of the decomposed load or store edges in the extended points-to graph instead of the corresponding SVFG edges ensures that this algorithm can expand all these pointers consistently even when SVFG does not contain edges connecting these pointers. After finding the pointers to expand, the algorithm recomputes the allocation size for each object allocation (node #1 and #2, not shown here) by walking the extended points-to graph, and the results converge.

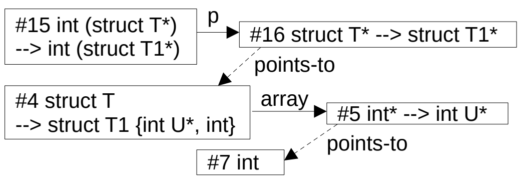


Figure 6: Subgraph for emitting types

Emitting types (step 4) with extended points-to graph. To determine the new types for objects and function prototypes, step 4 of the algorithm uses the subgraph with only cell nodes as a constraint graph and solves the new types for all cell nodes in the subgraph. Figure 6 shows the example subgraph continuing with the previous example. This subgraph preserves all the `contains` edges and `points-to` edges between the cell nodes, and these edges also describe the type dependency among the cells. Because the extended points-to graph merges all cell nodes whose type must be consistent, each of the cell nodes left can have type changes independently after the transform. In this example, initially, the pointer represented by node #5 will be expanded to carry uncompressed bounds (represented by `U*`). Because `struct T` represented by node #4 contains #5 as a member, this struct is modified to `struct T1` that includes the wider `array` pointer. This leads to the type change of node #16 to `struct T1*` and subsequently the change of node #15 to `int(struct T1*)`. With these new types solved, the subsequent instrumentation step can apply these type transforms to make space for the uncompressed bounds.

4 APPLYING MIFP TO CHERI

In this section, we present the implementation of MIFP based on the 64-bit RISCV implementation of CHERI [28]. The compiler toolchain is based on CHERI's fork of LLVM 13. The implementation targets CHERI's pure-capability mode, where there are no raw pointers in the application code. All pointers are fat pointers with compressed bounds before the transformation. We implemented the whole-program analysis and transform as a standalone tool using the SVF library [25], and we used scripts to integrate the transform with the compilation procedure. The prototype implementation supports C programs and works only for applications free of bad casts. We modified CHERI LLVM's RISCV backend to support generating new instructions for uncompressed bounds. We also modified the QEMU and CHERI's Flute processor core so that the instrumented programs can run on QEMU or FPGA.

4.1 ISA Extension

To reduce hardware costs, instead of doubling the maximal capability size, we represent the uncompressed bounds as a distinct capability-sized value that takes an extra register. For 128-bit CHERI capability on 64-bit RISC-V, the uncompressed bounds are also 128 bits in size. We added following instructions to utilize uncompressed bounds:

- csetboundsfull[imm]: Create uncompressed bounds from a base address and a bounds length value.
- cboundscheck: Perform bounds checks against uncompressed bounds.

CHERI has csetbounds and csetboundsimm instructions to create capability pointers with specific bounds, and the new instructions csetboundsfull and csetboundsfullimm work in parallel with them, except that the new instructions create uncompressed bounds. cboundscheck is a new instruction that takes an address (a base capability pointer and an immediate offset) and the uncompressed bounds as input for accurate bounds checking. We define multiple variants of cboundscheck instructions for checking different memory access sizes (1/2/4/8/16 bytes). If the size is irregular (e.g., because of operations like memset()), the compiler will break the check into two checks that check the first and last byte respectively. We use CHERI's existing capability load/store instructions to load/-store uncompressed bounds. All new instructions are available to user-level code, and no change to the privileged software is made.

To add the ISA extension to CHERI's Flute processor, we modified 91 lines of Bluespec SystemVerilog code in the CPU codebase. Most of the changes are in the decoder and the ALU. These changes take 1,121 extra LUTs (+1.6%) in the CPU. We find that the modifications do not affect the maximum clock speed as they are not on the critical path.

4.2 Compiler Support

In LLVM IR, we represent uncompressed bounds as separate values from the corresponding capabilities: for each capability pointer to type T (represented as T*), if it needs uncompressed bounds, we create another value void* as the uncompressed bound that propagates with T*. We add intrinsic functions for each new instruction that the whole-program transform can use to insert new instructions.

Extending Capabilities in Memory. When a capability pointer T* in memory needs uncompressed bounds, we replace the type of the pointer with a struct S{T*,void*} that contains both the original capability pointer T* and the uncompressed bounds void*. Loads and stores of capability pointers are extended to also load or store the uncompressed bounds. Any double pointers pointing to it (originally T**) are also modified to point to the struct instead, which becomes S*.

Extending Capabilities in Call and Returns. If a capability pointer is passed through a function call/return and it requires uncompressed bounds, we modify the function prototype and create extra arguments for the uncompressed bounds. If a capability pointer argument needs expansion, we create a new void* argument to pass the uncompressed bounds. If the return value is a capability pointer requiring expansion, we create a void** argument so that the callee can store the uncompressed bounds before returning, and the caller can load it after the return. Passing uncompressed bounds for variadic arguments is not supported in the prototype implementation.

4.3 Practical considerations

Besides the algorithm explained in Section 3, extra work is needed to make the transformed programs work. In this subsection, we describe fixes and workarounds for the problems we encountered.

Type size update. Changing the types of all values in the compiler IR alone is insufficient because size values derived from the types also need adjustment to make the program work. For example, dynamic allocators take size arguments for allocation, and functions like memset() and memcpy() also take size operands for memory manipulations. To transform the type of objects, we also need to update the arguments of these function calls. During the instrumentation, our prototype implementation will (1) use heuristics to identify such uses of type sizes, (2) analyze the size argument and decompose it into a linear form $Ax + C$ where A, C are constants and x is an unknown, and finally (3) replace the size with the new value $A'x + C'$ where $A' = A \times k$, $C' = C \times k$, and $k = \frac{\texttt{sizeof(DstTy)}}{\texttt{sizeof(SrcTy)}}$ is the type size ratio of the destination type and the source type. This heuristic can fail if the program uses the flexible array member feature in C, or if the allocation size is computed outside the function (e.g., the function is a wrapper for functions like malloc()), but it is sufficient to run all programs used in the evaluation.

Type metadata. Our prototype implementation uses the LLVM IR type system for cell nodes and expression nodes in the extended points-to graph, and this type system works well for the type-safe subset of C language. However, because compiler optimizations run before our algorithm, they can destroy the type information and create arbitrary casts, preventing the algorithm from correctly identifying object types. To work around this problem, we modified the Clang frontend to add additional metadata for the types of allocated objects and also added a transform to correct the types in LLVM IR before building extended points-to graphs.

Compatibility with external code. Although the transform takes the whole-program IR as input, there is always external code (likely library functions) that the transformed program has to be compatible with. When building the extended points-to graph, we implement an extra analysis that identifies all cells accessible by external code, and we prohibit type changes on these exposed cells. Also, because exposed cells are "roadblocks" that stop us from propagating uncompressed bounds from allocation sites (pointer sources) to downstream pointers along the value flow, we only need to compute bounds accuracy information for reachable pointers from allocation sites. Therefore, when running the forward dataflow analysis in step 2 of the transform (solving bounds accuracy), we stop the traversal when reaching an exposed pointer: only pointers that are safe to expand will have the bounds accuracy information and get checked for expansion. In our evaluation, the number of unsafe pointer uses that cannot result in pointer expansion is less than 10% of all unsafe uses in almost all programs.

Handling dead code. The whole-program IR can contain dead code, either from the source program or because of the function inlining. However, dead code can call other functions and pass pointers to them. In our implementation, some pointers in dead code will not have points-to targets (i.e., no outgoing points-to edges) because no code path would pass pointers to them, and these pointers and the transform would not impact each other. However, when we change the parameter or return value types of callee functions, the pointer types in the dead caller can be stale, creating errors when we transform all call sites for the callee. Our prototype implementation inserts syntactically correct code with runtime aborts at these code locations to pass the IR validation, and no such aborts get executed during our evaluation.

5 EVALUATION

We evaluate our MIFP implementation to measure how well it preserves the runtime and overhead reduction of bounds compression. We run all programs on our FPGA prototype with the modified Cheri-Flute soft core running CheriBSD (based on FreeBSD 14.0). We use Digilent Genesys 2 FPGA board, with a Kintex-7 XC7K325T-2FFG900C FPGA chip and 1GB of DDR3 RAM. The processor is single-core and runs at 25MHz.

5.1 Benchmark Programs

We choose 8 programs from Olden, 5 programs from Ptrdist, and the CoreMark benchmark as testing programs. Olden's `voronoi` is excluded because the program makes assumptions on data layout and the unmodified baseline does not work. Olden's `bh` is excluded because (1) the code relies on structural equivalence, which leads to bad casts in LLVM IR, and (2) our prototype currently does not support the downcasts used in `bh`. We modified Olden's `mst` to work around bad casts introduced by the LLVM SROA optimization. We modified CoreMark to (1) disable the CRC check on the memory block because it makes assumptions on memory organization and even the original CHERI version cannot pass the check, (2) remove the custom memory allocation which our prototype cannot analyze by replacing the allocation code with `malloc()` calls, and (3) remove unnecessary bad casts that break our analysis.

`coremark` contains benchmarks for linked list operations, and when modifying list node allocations in `core_list_init()`, we allocate all list nodes in a single `malloc()` call and all list data in another `malloc()`, so that the change needed to make the transform work is minimal. However, because the analysis cannot prove the pointer arithmetic in the linked list initialization to be safe[4], both the node pointer and the data pointer in the linked list get expanded. To make a fair comparison, we make another version `coremark-mifp` which allocates each linked list node and data block in distinct `malloc()` call so that the pointers in the linked list are not expanded. All statistics will include the results of both `coremark` and `coremark-mifp`.

We measure the runtime and memory usage of 3 versions of each program:[5] (1) `baseline`, where the program is compiled in CHERI pure-capability mode without running MIFP transform and all fat pointers use compressed bounds, (2) `mifp` version, where we run the transform to selectively use uncompressed bounds, and (3) `wide` version, where we expand all possible fat pointers to use uncompressed bounds regardless of bounds accuracy or whether the access is statically safe. The `baseline` version is the lower bound of overhead, and we use the `wide` version to estimate the upper bound of overhead if the bounds compression is disabled because of security concerns. The memory usage is taken from the maximum resident set size reported by `/usr/bin/time -al`. Because our implementation instruments an additional `cboundscheck` instruction for bounds checking without hoisting, the runtime increases of the `mifp` and `wide` versions are upper bound estimates, but we expect the memory usage increases to be accurate.

5.2 Correctness Verification

During the evaluation, the prototype implementation performs no pointer expansion in 8 out of our 14 benchmark programs; either because pointer uses are statically safe or the compressed bounds are already accurate. To better understand the program behavior and validate our analysis, we also create a statistics collection version of each program to (1) count the number of events listed in Table 1 (discussed in Section 5.3 below) and (2) perform runtime checks that compare the actual length and alignment of bounds against the maximal length and minimal alignment solved in step 2 of the algorithm to validate bounds accuracy. We use a test script to feed these programs with randomly generated input and run fuzzing tests in our QEMU implementation. After 277 total executions, 1680 out of 2061 instrumented runtime checks from all programs get executed at least once and no error is found. This confirms that the vast majority of pointers do not require uncompressed bounds, and our implementation is not missing pointers that should be expanded.

5.3 Performance Evaluation

Figure 7 shows the increase in run time and memory usage of `mifp` and `wide` versions of all programs. All overhead data is normalized against the `baseline` version.

To better explain the evaluation results, we show the statistics of all transformed programs in Table 1. The first three columns are collected at compile time and describe the extent of the changes that the instrumentation has made to the input programs. The number of pointer uses that are ignored (column 1) describes how the compatibility requirement with library code would impact the evaluation results. Because these numbers are small compared with the number of unsafe uses (column 2), we do not consider the compatibility requirement a threat to the validity of the results. The number of expanded pointer-type cell nodes (column 3) is an estimate of how many non-local pointer value declarations (including global variables and pointers in struct definitions and function prototypes) have the pointer type expanded. The last three columns are collected at run time and explain the performance numbers. The execution count of uncompressed bounds creation instructions (column 4) is an indicator of the memory overhead. The number of pointer loads and stores expanded (column 5) shows the memory access increase due to wider fat pointers, and the ratio of explicit bounds checks (column 6, the rightmost column) describes the percentage of bounds checks using uncompressed bounds after the transform.

5.3.1 Run Time Increase. Figure 7a shows that the normalized run time increases for all programs in the evaluation. The geomean is 8% for the `mifp` version and 35% for the `wide` version. This means

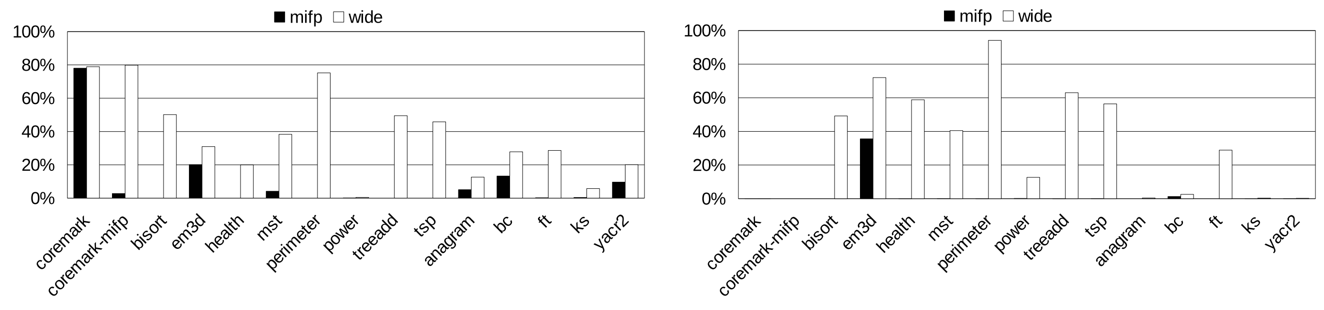
[4]In our prototype, LLVM Scalar Evolution cannot analyze IR values that are CHERI capability pointers.

[5]Because of the porting effort required, we cannot evaluate MIFP against related works, e.g. CCured.

Table 1: Statistics for benchmarks taking tranform

Benchmark	Ignored Uses[1a]	Unsafe Uses[1b]	Expanded Ptrs[2]	Bounds Creation [3]	Ptr Ld/St widened[4]	Full bounds checks[5]
coremark	7 (3%)	136 (53%)	12 (39%)	6	1.3×10^7 ($\approx$ 100%)	2.5×10^7 (65%)
coremark-mifp	7 (3%)	24 (9%)	6 (19%)	4	6.0×10^3 ($<$ 1%)	8.1×10^6 (32%)
em3d	5 (3%)	33 (22%)	6 (26%)	1.2×10^4	3.6×10^6 (15%)	2.3×10^7 (71%)
mst	1 (2%)	22 (37%)	6 (24%)	1.0×10^3	3.2×10^6 (19%)	6.6×10^6 (23%)
anagram	11 (4%)	19 (6%)	3 (8%)	4	5 ($<$ 1%)	7.6×10^5 (23%)
bc	61 (2%)	730 (29%)	49 (32%)	1.6×10^4	2.2×10^5 (48%)	6.9×10^5 (41%)
yacr2	1 ($<$ 1%)	444 (43%)	56 (78%)	85	2.0×10^6 ($\approx$ 100%)	3.6×10^7 (89%)

[1a,1b] Number of pointer uses that are (a) ignored because the pointer cannot be widened, and (b) considered unsafe during the analysis.
[2] Number of pointer-type cell nodes widened and their percentage among all pointer-type cell nodes in extended points-to graph.
[3] Execution count of `csetboundsfull[imm]` instruction, the number of uncompressed bounds created.
[4] Loads and stores for a pointer with uncompressed bounds versus the total number of pointer loads and stores, excluding stack spills.
[5] Number of explicit bounds checking with `cboundscheck` versus the total number of pointer dereferences in the program, excluding stack spills and accesses in library functions.



(a) Run time increase **(b) Memory usage increase**

Figure 7: Run time and memory usage increase of MIFP for all benchmarks

that MIFP's selective use of uncompressed bounds can reduce 79% of the run time increase compared with the unconditional use in `wide` versions.

Among these programs, the original `coremark` suffers the highest run time increase (78% for `mifp` and 79% for `wide`) because of frequent pointer loads and stores from linked list operations. With the change in linked list initialization, the run time increase of `coremark-mifp` reduces to 3%.

Besides `coremark`, both `em3d` and `bc` also incur non-negligible overhead. They have 20% and 13% run time increase in the `mifp` versions, and 31% and 28% increase in the `wide` versions, respectively. The `mifp` version of all other programs has less than 10% overhead. Except for `coremark` where the difference between `mifp` and `wide` is small, in all other programs, the analysis of MIFP can effectively reduce the runtime overhead of uncompressed bounds to a level near the baseline. Note that because `power` is floating-point heavy and only 8% of memory accesses are loading or storing pointers, it has a near-zero overhead even if we expand all possible pointers in the `wide` version.

5.3.2 Memory Usage Increase. Figure 7b shows the normalized memory usage increases for all programs. The `mifp` versions incur 36% memory overhead in `em3d`, 1% overhead in `bc`, and near-zero overhead for the rest of the programs, therefore the geomean is 2%. The geomean of the memory overhead for the `wide` version is 28%. While having a large performance overhead, `coremark` has near-zero memory overhead for all versions because the original sizes of the data structures (linked list, matrix, etc) are fixed at around 2KB, which is small compared with the total memory usage (about 2MB). Overall, MIFP is also effective in avoiding the memory overhead introduced by uncompressed bounds; the memory overhead is only about 7% of the full `wide` pointer expansion overhead.

6 DISCUSSION

In this section, we discuss (1) how to support programs using unions or arbitrary pointer casts, (2) the additional limitations MIFP and how they can be mitigated, and (3) the potential of MIFP for other purposes outside accurate bounds checking.

Type and memory layout. Because our prototype uses LLVM IR types to represent object memory layouts, unions and arbitrary casts on pointers are hard to support because they introduce conflicting types not representable in the extended points-to graph. In addition, unions and arbitrary casts can threaten the accuracy of the pointer analyses MIFP depends on, challenging the correctness of the value flow graph and the points-to graph. In the prototype implementation, pointers to aggregate types (structs and arrays), functions, or other pointers can only be cast from or to `void*` or `char*`. Pointers to elementary data types (`int`, `double`, etc) can cast to each other but not to the listed pointer types above. Unions containing pointers or aggregate types are not supported. Pointer-to-integer conversion is supported but integer-to-pointer conversion is not. To support programs using these features, one can replace the type information of cell nodes in the extended points-to graph with data structures that better model object memory layouts and reimplement the algorithms to operate on the memory layout instead of types. We note that arbitrary casts are generally not supported in safe programming languages. This includes new memory-safe languages like Rust and Go and prototype C dialects that use the type system to guide check instrumentation [4, 7, 10, 33]. The programmer needs to either rewrite the code to avoid the casts or wrap the code in an unsafe region.

Handling `sizeof()`. MIFP modifies data layout according to whole-program analysis results, effectively making the `sizeof()` values *link-time constants.* In our prototype implementation, `sizeof()` is still expanded early in the compilation pipeline. Therefore, our implementation cannot identify all references of `sizeof()` perfectly. One solution is to implement MIFP transform as a source-to-source transform so that `sizeof()` is lowered after the transform, which requires significant effort in modifying the MIFP implementation and the depending analyses. Another workaround is to develop a checker in the language frontend to catch uses of `sizeof()` that cannot be recognized later in the transform, and manually port existing code to remove these unsupported uses. We expect that most uses of `sizeof()` are located at allocation sites. `sizeof()` uses in programming paradigms like getting array length with `sizeof(array)/sizeof(array[0])` can work without change.

Library Code. Although MIFP can instrument programs without human effort, it cannot instrument allocations not included in the whole-program IR, especially allocations from library functions. Therefore, pointer arithmetic on pointers from libraries cannot use uncompressed bounds for checking. This is a fundamental limitation of our approach. However, we note that best practices of library API designs typically discourage exposing internal data structures to application code, for example, opaque pointers are preferred. Therefore, following such best practices can reduce the impact of library code.

Atomicity considerations. In our implementation, the uncompressed bounds and the corresponding capability pointer are treated as separate values, so two load/store instructions are needed to access both of them. This can create problems for applications that assume pointer load/stores are atomic operations. To fix this problem without source code changes, hardware support for atomic access of these fat pointers is required. One implementation strategy is to expand the maximum fat pointer size so that the entire fat pointer with uncompressed bounds can be transferred between memory and register with a single instruction.

Wild out-of-bounds pointers. One benefit of MIFP's approach is the potential to support wild out-of-bounds pointers. When a pointer is moved too far away from the bounds, CHERI will not be able to represent the compressed bounds and will invalidate the pointer. If supporting such pointers is required, the only option is carrying the uncompressed bounds. This is not possible with mitigation techniques based on memory padding.

Generalization for other pointer metadata. MIFP uses the extended points-to graph to determine data type transforms required for spatial memory safety. However, we expect the graph representation to also work for other security policies that attach metadata on a per-pointer basis. For example, to reduce the cost of catching use-after-free or double-frees, a fine-grained scheme can use compiler analysis to find all pointers that may trigger such errors and use the graph representation to only create metadata for these vulnerable pointers without affecting other safe pointers [15, 33]. In application domains where the atomicity problem caused by the extra metadata is insignificant (e.g., on embedded systems where there is only one thread), it is also possible to enforce multiple security policies, each with its own per-pointer metadata.

7 CONCLUSION

In this paper, we introduced MIFP and showed how its whole-program transform selectively adds uncompressed bounds for unsafe pointers. The extended points-to graph makes it possible for MIFP to select fat pointer representations on a per-pointer basis and only attach uncompressed bounds for necessary fat pointers. The evaluation shows that MIFP is effective in maintaining the performance and memory savings of bounds compression while avoiding bounds inaccuracy.

We believe that fat pointer with bounds compression is a promising practical solution against spatial memory errors, and we consider MIFP as one necessary option for fat-pointer schemes to be deployed in security-sensitive domains. In the future, we also envision MIFP's method being adapted for other security-related problems that benefit from per-pointer metadata.

ACKNOWLEDGMENTS

We would like to thank Ruslan Bukin and Robert N. M. Watson from University of Cambridge for their technical support on CHERI. This research is supported in part by a contract with Huawei Canada, NSERC CRD Grant CRDPJ 541942-19 and a Tier 1 Canada Research Chair.

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the USENIX security symposium (USENIX Security '09)*. Montreal, Canada. https://www.usenix.org/legacy/event/sec09/tech/full_papers/sec09_memory.pdf

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

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

[4] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *Programming Languages and Systems*, Rocco De Nicola (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 520–535.

[5] Sourav Das, R. Harikrishnan Unnithan, Arjun Menon, Chester Rebeiro, and Kamakoti Veezhinathan. 2019. SHAKTI-MS: A RISC-V Processor for Memory Safety in C. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*. Phoenix, AZ, USA, 19–32. https://doi.org/10.1145/3316482.3326356

[6] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming

Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. Seattle, WA, USA, 103–114. https://doi.org/10.1145/1346281.1346295

[7] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60. https://doi.org/10.1109/SecDev.2018.00015

[8] Robert Hundt, Sandya Mannarswamy, and Dhruva Chakrabarti. 2006. Practical structure layout optimization and advice. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 12 pp.–244. https://doi.org/10.1109/CGO.2006.29

[9] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. 2021. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 916–929. https://doi.org/10.1109/ISCA52012.2021.00076

[10] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: a safe dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[11] Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. *Security Analysis of CHERI ISA*. Technical Report. Microsoft Security Response Center. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf [Accessed: 2022-06-26].

[12] Tina Jung, Fabian Ritter, and Sebastian Hack. 2021. PICO: A Presburger In-Bounds Check Optimization for Compiler-Based Memory Safety Instrumentations. *ACM Trans. Archit. Code Optim.* 18, 4, Article 45 (jul 2021), 27 pages. https://doi.org/10.1145/3460434

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

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

[15] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. 2022. C to Checked C by 3c. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 78 (apr 2022), 29 pages. https://doi.org/10.1145/3527322

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

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

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

[19] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. *SIGPLAN Not.* 37, 1 (jan 2002), 128–139. https://doi.org/10.1145/565816.503286

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

[21] Probir Roy and Xu Liu. 2016. StructSlim: A lightweight profiler to guide structure splitting. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 36–46.

[22] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. 2022. HeapCheck: Low-Cost Hardware Support for Memory Safety. *ACM Trans. Archit. Code Optim.* 19, 1, Article 10 (jan 2022), 24 pages. https://doi.org/10.1145/3495152

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

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

[25] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

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

[27] Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned Memory Models for Program Analysis. In *Verification, Model Checking, and Abstract Interpretation*, Ahmed Bouajjani and David Monniaux (Eds.). Springer International Publishing, Cham, 539–558.

[28] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037

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

[30] Shengjie Xu, Wei Huang, and David Lie. 2021. In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Granularity Protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 224–240. https://doi.org/10.1145/3445814.3446761

[31] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 88–99. https://doi.org/10.1109/ISSRE.2014.20

[32] Chao Yu, Probir Roy, Yuebin Bai, Hailong Yang, and Xu Liu. 2018. LWPTool: A lightweight profiler to guide data layout optimization. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (2018), 2489–2502. https://doi.org/10.1109/TPDS.2018.2840992

[33] Jie Zhou, John Criswell, and Michael Hicks. 2022. Fat Pointers for Temporal Memory Safety of C. https://arxiv.org/abs/2208.12900. https://doi.org/10.48550/ARXIV.2208.12900

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