FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications

Myoung Jin Nam
mjnam@formal.korea.ac.kr
Korea University
Seoul, South Korea

Periklis Akritidis
akritid@niometrics.com
Niometrics
Singapore

David J Greaves
David.Greaves@cl.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

ABSTRACT

Security mechanisms for systems programming languages, such as fine-grained memory protection for C/C++, authorize operations at runtime using access rights associated with objects and pointers. The cost of such fine-grained capability-based security models is dominated by metadata updates and lookups, making efficient metadata management the key for minimizing performance impact. Existing approaches reduce metadata management overheads by sacrificing precision, breaking binary compatibility by changing object memory layout, or wasting space with excessive alignment or large shadow memory spaces.

We propose FRAMER, a capability framework with object granularity. Its sound and deterministic per-object metadata management mechanism enables direct access to metadata by calculating their location from a *tagged pointer* to the object and a compact supplementary table. This may improve the performance of memory safety, type safety, thread safety and garbage collection, or any solution that needs to map pointers to metadata. FRAMER improves over previous solutions by simultaneously (1) providing a novel encoding that derives the location of per-object metadata with low memory overhead and without any assumption of objects' alignment or size, (2) offering flexibility in metadata placement and size, (3) saving space by removing any padding or re-alignment, and (4) avoiding internal object memory layout changes. We evaluate FRAMER with a use case on memory safety.

CCS CONCEPTS

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

KEYWORDS

Security, Tagged Pointers, Memory Safety, Bounds Checking, Object-Capability Model, LLVM

1 INTRODUCTION

Despite advances in software defenses, exploitation of systems code written in unsafe languages such as C and C++ is still possible. Security exploits use memory safety vulnerabilities to corrupt or leak sensitive data, and hijack a vulnerable program's logic. In response, several defenses have been proposed for making software exploitation hard.

Current defenses fall in two basic categories: those that let memory corruption happen, but harden the program to prevent exploitation, and those that try to detect and block memory corruption in the first place. In the first category, for instance, Control-flow Integrity (CFI) [1, 14, 20, 35, 37, 45–47, 53–55] contains all control flows in a statically computed Control-flow Graph (CFG), while Address Space Layout Randomization (ASLR) hides the available CFG when the process executes. Both approaches can be bypassed [15, 42], since memory corruption is still possible, albeit exploitation is much harder.

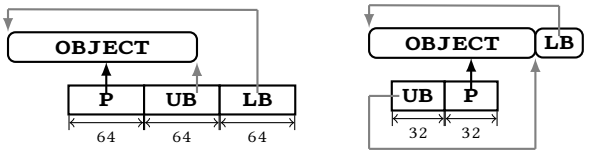
We focus on the second category, including deterministic approaches that detect and block memory safety violations by maintaining runtime metadata for access rights and instrumenting the program to block unintended accesses at runtime [3, 5, 13, 16, 19, 21–23, 32–34, 52]. These systems can offer deterministic guarantees by preventing memory corruption in the first place, however tracking all objects (or pointers) incurs heavy performance overheads. Performance is critical for adoption since unsafe languages like C/C++ are employed for performance-sensitive applications. Some of these systems trade accuracy for speed by allowing false negatives, and hence are more useful for troubleshooting than security.

Some existing techniques trade off *compatibility* for high locality of reference, however, it is desirable to minimise the disruption owing to tacit assumptions by programmers and compatibility with existing code or libraries that cannot be recompiled. In particular, so-called *fat pointers* [34] impose incompatibility issues with external modules, especially precompiled libraries.

With these limitations in mind, *object-capability models* [9, 25, 48, 49] using hardware-supported tags become very attractive, because they can manage compatibility and control runtime costs. However, they cannot entirely avoid undesirable overheads such as metadata management related memory accesses just by virtue of being hardware-based. In turn, some hardware-based solutions also trade accuracy for acceptable performance [**?**].

In this paper, we present FRAMER, a memory-efficient capability model using *tagged pointers* for fast and flexible metadata access. FRAMER provides efficient per-object metadata management that enables direct access to metadata by calculating their location using the (currently) unused top 16 bits of a 64-bit pointer to the object and a compact supplementary table. The key considerations behind FRAMER are as follows.

Firstly, FRAMER enables the memory manager freedom to place metadata in the associated header near the object to maximise spatial locality, which has positive effects at all levels of the memory hierarchy. Headers can vary in size, unlike approaches that store the header at a system-wide fixed offset from the object, which may be useful in some applications. Headers can also be shared over object instances (although we do not develop that aspect in this paper). Our evaluation shows excellent D-cache performance where the performance impact of software checking is, to a fair extent, mitigated by improved instructions per cycle (IPC).



(a) Fat pointers (b) SGXBounds

Figure 1: Embedded Metadata: P, UB, and LB represent a pointer itself, upper bound, and lower bound, respectively.

Secondly, the address of the header holding metadata is derived from tagged pointers regardless of objects' alignment or size. We use a novel technique to encode the *relative location* of the header in unused bits at the top of a pointer. Moreover, the encoding is such that, despite being relative to the address in the pointer, the tag does not require updating when the address in the pointer changes. A supplementary table is used only for cases where the location information cannot be directly addressed with the additional 16-bits in the pointer. The address of the corresponding entry in the table is also calculated from our tagged pointer. With the help of the tag, this table is significantly smaller compared to typical *shadow memory* implementations.

Thirdly, we avoid wasting memory from any padding and superfluous alignment, whereas existing approaches using shadow space [3, 17, 25, 32, 39] re-align or group objects to avoid conflicts in entries, FRAMER provides great flexibility in alignment, that completely removes constraining the objects or memory. The average of space overheads of our approach is 20% for full checking despite the generous size of metadata and the supplementary table in our current design.

Fourthly, our approach facilitates *compatibility*. Our tag is encoded in otherwise unused bits at the top of a pointer, but the pointer size is unchanged and contiguity can be ensured.

The contributions of this paper are the following:

- We present an efficient encoding technique for relative offsets that is compact and avoids imposing object alignment or size constraints. Moreover, it is favourable for hardware implementation.
- Based on the proposed encoding, we design, implement and evaluate FRAMER, a generic framework for fast and practical object-metadata management with potential applications in memory safety, type safety and garbage collection.
- We illustrate the use of FRAMER with a case study on spatial memory safety that guarantees near-zero false negatives/-positives, using our framework to allow inexpensive validation of pointer dereferences by associating pointers to object metadata containing bounds information. We demonstrate promising low memory overheads and high instruction-level parallelism.

2 BACKGROUND AND RELATED WORK

Several approaches have been proposed for tracking memory and detecting memory-related errors. The overhead is one of the biggest challenges of run-time protection mechanisms along with detection coverage. In light of recent hardware-based solutions, memory bandwidth remains the main performance constraint for a broad class of applications and additional cache misses or DRAM row activations, owing to lack of spatial consistency, larger footprint or poor structure alignment, always needs to be minimised. We review here trade-offs of systems that either track objects or pointers.

Pointer-based tracking guarantees near *complete memory safety*. Its per-pointer metadata hold the valid range that a pointer is allowed to point to. This enables it to detect internal overflows easier, such as an array out-of-bounds inside a structure, unlike object-based approaches.

Pointer-based approaches are often implemented using *fat pointers* [5, 34**?**]. They define a new pointer representation that carries metadata with itself, thus increasing locality but sacrificing *compatibility*. Since fat pointers increase the number of bytes used to hold a pointer (Fig. 1a) they require modification of the memory layout and this damages compatibility with non-instrumented code. Moreover, updates to fat pointers spanning multiple words are not atomic, while some parallel programs rely on this.

Several pointer-based approaches [32] chose memory layout compatibility over locality. Using *disjoint metadata* achieves compatibility by storing metadata in a separate memory region. Intel MPX [19, 31, 36] is an ISA extension that provides hardware-accelerated pointer-checking using disjoint metadata in a bounds table holding per-pointer metadata as illustrated in Fig. 2b.

Pointer-tracking approaches' strong guarantees comes with the additional runtime overhead from metadata copy and update at pointer assignment, while object-based approaches update metadata only at memory allocation/release. In addition, the number of pointers is typically larger than that of allocated objects, so pointer-intensive programs may suffer from heavier runtime overheads.

Hardware support [10, 19, 25, 49**? ?**] does not remove this overhead. Reportedly, MPX suffers due to lack of memory even with small working sets [24], and has turned out to be slow for pointer-intensive programs, owing to exhausting the limited number of special-purpose bounds registers (4 registers), requiring spill operations from regions of memory that themselves require management and consume D-cache bandwidth and capacity.

Due to the heavy cost of per-pointer metadata, more techniques track objects. *Object-based* approaches [22**? ?**] store metadata per object and also make a trade-off against complete memory safety. By not changing the memory layout of objects, they offer compatibility with current source and precompiled legacy libraries.

Modern approaches reduce slowdown using a *shadow space* that allows direct array access to metadata [3, 8, 10, 17, 32, 34, 38, 51]. Beyond early techniques' *byte-to-byte* mapping of the application space, recent techniques reduced the size of shadow space with compact encoding, at the cost of minimum allocation size or loss of some precision. An example is Baggy bounds checking (BBC) [3]. BBC mandates object alignment to the base of a block, to prevent metadata conflicts caused by multiple objects in one block. In addition, it pads each object to the next power of two, so that each one-byte sized entry stores only log_2(padded object size). BBC performs *approximate* bounds checking, tolerating going out-of-bounds yet within the padded bound.

Address Sanitizer [39] (ASan) utilizes shadow space differently. It re-aligns and pads each object with *redzones* front and back as shown in Fig. 2a, and considers access to redzones as out-of-bounds.

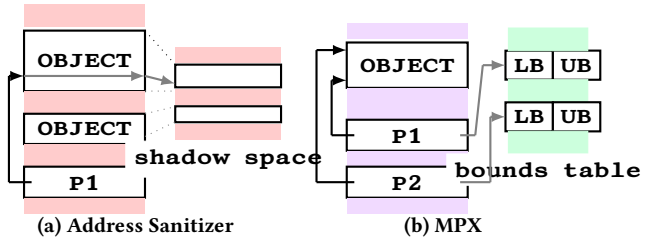


Figure 2: Disjoint Metadata

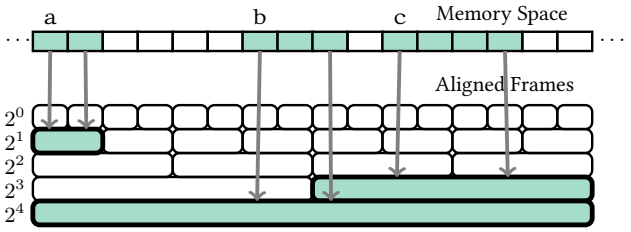


Figure 3: Aligned frames in memory space: a memory space can be divided into frames that are defined by memory blocks that are 2^n-sized and aligned by their size. A memory object's wrapper frame is the smallest frame completely containing the object. For instance, the 2-byte sized object a's wrapper frame size is 2^1 (called 1-frame). In the same way, objects b and c's wrapper frames are 4-frame and 3-frame, respectively.

At memory access, ASan derives the address of its corresponding entry from a pointer, and the entry tells if the address is *addressable*. Disadvantage of ASan is that its error detection relies on spatial *distance*. It loses track of pointers going far beyond of redzone and reaching another object's valid range, so fails to address tracking *intended referents* [22]. The wider the redzone, the more errors ASan detects. ASan detects most errors, but it is less deterministic in theory and trades-off memory space for detection coverage. In our experiments, ASan and FRAMER's normalised memory footprints are 8.84 and 1.23, respectively.

Rather than fat pointers or shadow space, *tagged pointers* [23, 24] can instead be used. SGXBounds [24] trades-off address space for speed and near-complete memory safety. SGXBounds makes objects carry their metadata in a *footer* as shown in Fig. 1b, and utilizes the higher 32 bits of a pointer to hold the metadata location (upper bound of its referent at the same time). Storing the absolute address of bounds frees SGXBounds from false negatives that challenge many object-tracking approaches. This approach works when there are enough spare bits in pointers, which is the case with SGX enclaves, where only 36 bits of virtual address space are currently supported.

Hardware accelerated tagged pointers are available without sacrificing address space. ARM v8.5 ISA [26, 27] introduces the Memory Tagging Extension (MTE) assigning a 4-bit tag to each 16 bytes at memory allocation, and tapping memory accesses with incorrect tags in the pointer. The memory bandwidth impact will depend greatly on the underlying hardware architecture and could be close to zero if the tags are largely implemented in separate hardware resources and blocks are normally cleared on allocation. However, this approach has 1/16 chance of false negatives at each memory access.

In this work, to achieve deterministic memory protection with data memory efficiency, while preserving the full 48-bit address space available in contemporary CPUs, we sacrifice dynamic instruction counts. We (1) rein in the increase in extra cache misses for metadata (owing to spatial locality compared with a total shadow memory approach) and (2) we tolerate an increase in outgrowth of executed instructions for arithmetic operations. This may sound unfavourable, but note that we can move to an even sweeter spot in the future where the instruction overhead for calculation is reduced via customised ISA. Our framework provides a novel encoding that derives metadata location from a tagged pointer, lowering both memory footprint and cache misses. In our experiments, normalised L1 D-cache miss counts for FRAMER and ASan on average are 1.40 and 2.31, respectively.

FRAMER can be the base of a solution for both (1) practical deployment with customised ISA for its efficiency of memory footprint and cache memory and (2) sound runtime verification during development.

3 FRAMER APPROACH

In a nutshell, FRAMER places per-object metadata close to their object and calculates the location of metadata from only (1) an inbound pointer and (2) additional information tagged in the otherwise unused, top 16 bits of the pointer. We exploit the fact that *relative addresses* can be encoded in far fewer bits than *absolute* addresses with assistance from the memory manager to restrict the distance between the allocation for an object and a separate object for its metadata. In our case, the metadata can be stored in front of the object, essentially as a *header* that an object carries with itself, requiring only a single memory manager allocation. For the remaining cases where the relative address cannot fit in a 16-bit tag, we use a compact supplementary table to locate the header. The tag encodes when this is the case, and also sufficient information to locate the supplementary entry.

We are now going to introduce the concept of *frames* used to encode relative offsets. We first define frames in Section 3.1 and show how to calculate an object's *wrapper frame* in Section 3.2. In Section 3.3 we explain how relative location can be encoded in a tagged pointer using these concepts, and how to exploit this encoding to reduce the supplementary table's size.

3.1 Frame Definitions

To record the relative location in the top 16 bits of a 64-bit pointer, which are spare in contemporary CPUs, we define a logical structure over the whole data space of a process, including statics, stack, and heap. The FRAMER structures are based on the concept of *frames*, defined as memory blocks that are 2^n-sized and aligned by their size, where n is a non-negative integer. A frame of size 2^n is called *n-frame*. A memory object x will intrinsically lie inside at least one bounding frame, and x's *wrapper frame* is defined as the smallest frame completely containing x, so there exists only one wrapper

frame for *x*. For instance, in Fig. 3, each sharp-cornered box represents a byte, and contiguous coloured bytes are objects allocated in memory (e.g. object a has a size of 2 bytes). Memory space is divided to frames illustrated as round-cornered boxes. Objects a , b and c's wrapper frames are $(n = 1)$-frame (or 1-frame), 4-frame, and 3-frame, respectively. For $0 \leq m < n$, we call m-frames placed inside an n-frame f, f's *subframes*.

Frames have several interesting properties. Firstly, an n-frame is aligned by 2^m for all $m < n$. Secondly, an object's wrapper frame size is not proportional to the object's size. As shown in Fig. 3, the object b has a larger wrapper frame than c, even though b's size is smaller. This is because the wrapper frame size for an object is determined by both the object's size and location. Thirdly, as discussed previously, an object's wrapper frame is defined as the smallest frame containing the object. Given an object *x*, its wrapper frame is obtained by finding a frame having *x*'s base (i.e. lower bound) and upper bound in its lower-addressed $(n-1)$-subframe and higher-addressed $(n-1)$-subframe, respectively. For example, in Fig. 3, object b's lower and upper bound are placed in b's wrapper frame (4-frame)'s lower-addressed and higher-addressed 3-subframes, respectively. It is trivial to prove that an object's wrapper frame is the frame having the object's lower and upper bound in its biggest subframes, as presented in Appendix A.1.

Following basic malloc semantics, FRAMER does not natively support object movement or growth (we reset its wrapper frame at realloc). Therefore, there exists a unique wrapper frame for each object, and it is determined at memory allocation. Since it does not change during the life time of an object, we can encode the metadata location using an offset relative to the wrapper frame. At memory allocation, we determine the wrapper frame for the allocated object and store the metadata offset in the pointer tag.

3.2 Frame Selection

We now show how to calculate the size of the wrapper frame, given an object. We call an object whose wrapper frame is an n-frame an *n-object*. For any k-object *o*, since its wrapper frame (i.e. a k-frame) is aligned by 2^k by definition, the addresses of all bytes in the frame coincide in their most significant $(64-k)$ bits, and so do the addresses of all bytes in *o*. In addition, the base and upper bounds are located in the lower and higher-addressed $(k-1)$-frame, respectively. This means that the $(k-1)^{\text{th}}$ least significant bit of the base and that of the upper bound are complementary to each other.

Based on these, we can calculate k, the binary logarithm ($\log_2$) of *o*'s wrapper frame's size. Let $(b_{63}, ..., b_1, b_0)$ and $(e_{63}, ..., e_1, e_0)$ bit vectors of k-object *o*'s base and upper bound respectively, and X a *don't care* value. We derive $\log_2$(wrapper frame size) by performing XOR (exclusive OR) and CLZL (count leading zeros) operations as follows (b_{63} is the most significant):

$(b_{63},$	...,	$b_k,$	$b_{(k-1)},$	$b_{(k-2)},$	...,	$b_0)$	
$(e_{63},$	...,	$e_k,$	$e_{(k-1)},$	$e_{(k-2)},$	...,	$e_0)$	XOR
$(0,$	...,	$0,$	$1,$	X,	...,	X)	CLZL
						$(64-k)$	

We then get k by subtracting the result of the CLZL operation from 64, since $k = 64 - (64 - k)$.

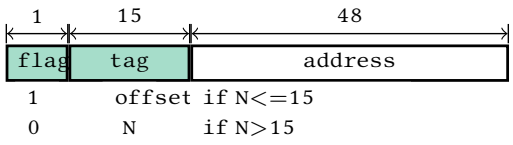


Figure 4: Tagged pointer: the tag depends on the value of N (binary logarithm of the wrapper frame size of a referent object).

3.3 Metadata Storage Management

FRAMER's memory manager places metadata in a *header* before the object contents. For instance, in Fig. 5, a , b and c are all objects containing a header. Using any bounding frame as a frame of reference, we can encode the location of the object's metadata (i.e. header) relative to the base of this frame. We can then derive the metadata location given an inbound pointer using the following:

(1) the binary logarithm of the bounding frame size ($N = log_2 2^N$)
(2) an offset to a header from the bounding frame base

Given an inbound pointer and a bounding N-frame, aligned by 2^N by definition, we derive the bounding frame's base by clearing the pointer's N least significant bits. This means that once a bounding frame's N value is known to us, we can obtain the frame's base without any other information but the address in an inbound pointer's 48 lower bits.

Having the value of N at hand, we may tag pointers with the offset from the bounding N-frame's base to the header. However, even with the value of N provided, the 16 bits of the tag cannot hold the large offsets required for some combinations of wrapper frame size and header location. For instance, a $(N = 20)$-object's offset (20-frame's base $\sim$ the header) may need up to 19 bits.

To stuff the limited space of unused 16 bits of a pointer with both an arbitrary offset and N value, FRAMER divides the virtual address space into *slots* with a fixed size of 2^{15} bytes, aligned to their size, i.e., 15-frames. Slots are set to a size of 2^{15} so that offsets to the header of objects can be encoded in the unused 15 bits of a pointer (one bit among 16 is reserved for a flag described subsequently). In Fig. 5, d_a is the offset to the header of the object a.

FRAMER then distinguishes between two kinds of objects, depending on their wrapper frame size, namely *small-framed* and *large-framed* objects. Small-framed objects are defined as $(N \leq 15)$-objects, i.e. objects whose wrapper frame size is less than/equal to 2^{15}. Large-framed objects are defined as $(N > 15)$-objects. For example, in Fig. 5, object a is small-framed, whereas b and c are large-framed. One extra bit, in particular the most significant, is used for a *flag* indicating if the object is small-framed or large-framed as shown in Fig. 4. We handle objects differently depending on their kind.

3.3.1 Small-framed Objects. Small-framed objects are completely contained in a single slot, so any pointer to them is derived to the slot base by zeroing the 15 least significant bits of the pointer. The offset of a small-framed object x's header from the base of the slot containing x is stored in the 15 bit pointer tag. For instance, in Fig. 5

we tag pointers to the small-framed object a with d_a (slot0's base $\sim$ a's header).

We further turn on the most significant bit of the pointer to indicate that the particular object is small-framed. FRAMER then recognises a pointer to a small-framed object by the flag being on and takes the 15-bit tag as an offset to its header from the base of the slot containing the object. This way, we avoid storing the value of N for small-framed objects.

In summary, when we retrieve metadata from a header of a small-framed object (i.e., flag is on), inbound (in-slot) pointers are derived to the base of the slot by zeroing the 15 least significant bits ($log_2(slot\_size) = 15$), and then to the address of the header by adding the offset to the base address of the slot as follows:

```
// FLAGMASK: (1ULL ¡¡ 63)
// flag is on

offset = (taggedptr & FLAGMASK) ¿¿ 48;
slotbase = untaggedptr & (0ULL ¡¡ 15);
headeraddr = slotbase + offset;
objbase = headeraddr + headersize;
```

Small-framed objects are overwhelmingly common. Our experiments showed the number of large-framed objects is very low compared to small-framed ones: 1: $>$ 200,000 on average and 1: millions in some benchmarks. This is fortunate, because the header location for small-framed objects is derived from tagged pointers alone, while large-framed objects require additional bits of information. These additional bits are provided by entries in a supplementary table. We stress here that the location of this entry is also derived using the tag in a way that enables much smaller tables than typical shadow memory implementations. We describe this encoding next.

3.3.2 Large-framed Objects. Since large-framed objects span several slots, zeroing the 15 least significant bits (log_2 of slot size) of a pointer does not always lead to a unique slot base, thus the offset in the tag cannot be solely used to derive their relative location. In Fig. 5, a pointer to a 16-object b can derive two different slot bases (slot0 and slot1) depending on the pointer's value, and that is the case for 17-object c (slot1 and slot2). In addition, the offsets from the base of their wrapper frame ($(N > 15)$-frame) to an $(N > 15)$-object's header may not fit in spare bits. Hence, for large-framed objects, we need to store additional location information in our supplementary table, and use a different encoding in the pointer tag to derive the address of the corresponding entry from any pointer to the object.

During program initialisation, we create a table holding an entry for each 16-frame. We call such a frame a *division*. Each entry contains one sub-array and the sub-array per division is called a *division array*. Each division array contains a fixed number of entries potentially pointing to metadata headers, in the current implementation as follows:

```
typedef struct ShadowTableEntryT –
  HeaderTy *divisionarray[48]; // 64-16
 DivisionT;
```

Contrary to small-framed objects, in the tag for large-framed objects we store the binary logarithm of their wrapper frame size (i.e., $N = log_2 2^N$) as shown in Fig. 4. The address of an entry in a

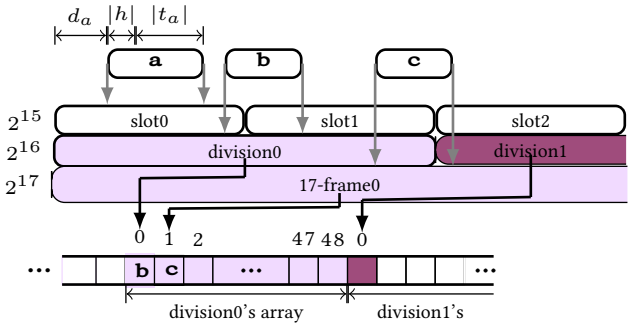


Figure 5: Access to division array: the object a is small-framed, while b and c are large-framed. d_a is the offset to a. h denotes a header and $|t_a|$ is the size of a. b and c's entries are mapped to the same division array. The entries in the division arrays store their corresponding object's header location, while the small-framed object a does not have an entry. Only one entry of division1's array is actually used, since the division is not aligned by 2^{17}.

division array is then calculated from an inbound pointer and the N value, and the entry holds the address of a header. By definition, a wrapper frame of an $(N \geq 16)$-object is aligned by its size, 2^N, therefore, the frame is also aligned by 2^{16}. This implies that a $(N \geq 16)$-frame shares the base address with a certain division, and is mapped to that division.

Each $(N \geq 16)$-object maps to one division array, but that division array contains entries for multiple large-framed objects. In Fig. 5, both division0 and 17-frame0 are mapped to division0. Their mapped division (division0) is aligned by 2^{17} at minimum, while division1 is aligned by 2^{16} at max.

The tag N can be used as an index into the division array to associate a header pointer, stored in an entry in the division array, with each large-framed object mapped to the same division. For each $N \geq 16$, at most one N-object is mapped to one division array, and the proof is presented in Appendix A.2. We use the value N as an index of a division array, and tag N in the pointer. Given a N value-tagged pointer (flag==0), we derive the address of an entry as follows:

```
// UBASE: division base of userspace's base
// SCALE: binary logarithm of divisionsize, i.e
    . 16
// TABLE: address of a supplementary table
// flag is off

framebase = p & (0ULL ¡¡ N); // p is assumed
    untagged here
tableindex = (framebase - UBASE) / (1ULL ¡¡
    SCALE);
DivisionT *M = TABLE + tableindex;
headeraddr = M-¿divisionarray[N - SCALE];
```

The base of the wrapper frame (i.e. the base of the division) is obtained by zeroing the least significant N bits of the pointer. The address of its division array is then derived from the distance from

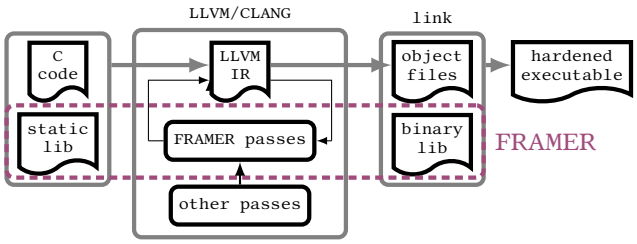


Figure 6: Overall architecture of FRAMER

the base of virtual address space and $log_2(division\_size)$ (2^{16}). Finally we access the corresponding entry with the index N in the division array.

Entries in a division array may not always be used, since an entry corresponds to one large-framed object, which is not necessarily allocated at any given time, e.g. if object `b` is not allocated in the space in Fig. 5, 0th element of `division0`'s array would be empty. This feature is used for detecting some dangling pointers, and more details are explained in Section 5.2.

Unlike existing approaches using shadow space, FRAMER does not re-align objects to avoid conflicts in entries. Our *wrapper frame-to-entry* mapping allows wrapper frames to be overlapped, that gives full flexibility to memory manager.

We could use different forms of a header such as a *remote* header or a *shared* header for multiple objects, with considering a cache line, stack frame, or page. In addition, although we fixed the division size (2^{16}), future designs may offer better flexibility in size.

We showed how to directly access per-object metadata only with a tagged pointer. Our approach gives great flexibility to associate metadata with each object; gives full freedom to arrange objects in memory space, that removes padding objects unlike existing approaches using shadow space. This mechanism can be exploited for other purposes: the metadata can hold any per-object data.

4 FRAMER IMPLEMENTATION

This section describes the current implementation of FRAMER which is largely built using LLVM. Additionally, we discuss how we offer compatibility with existing code.

4.1 Overview

There are three main parts to our implementation: FRAMER LLVM passes, and the static library (lib), and the binary lib in the dashed-lined box in Fig. 6. The target C source code and our hooks' functions in the static lib are first compiled to LLVM intermediate representation (IR). Our main transformation pass instruments memory allocation/release, access, or optionally pointer arithmetic in the target code in IR. In general, instrumentation simply inserts a call to lib functions, however, our use of header-attached objects and tagged pointers requires more transformation at compile-time. The third part is wrappers around `malloc` family routines and string functions. Our customised compiler optimisations are discussed in Section 6.

We also had to modify the LLVM framework slightly. Our main transformation is implemented as a LLVM Link Time Optimisation (LTO) pass for whole program analysis, and runs as a LTO pass on gold linker [30], however, incremental compilation is also possible.

We also insert a prologue that is performed on program startup. The prologue reserves address space for the supplementary metadata table, but pages are only allocated on demand.

4.2 Memory Allocation Transformations

We instrument memory allocation and deallocation to prepend headers and update metadata by transforming the target IR code at compile time.

4.2.1 Stack-allocated Objects (address-taken locals). For each local allocation of aggregate-type that needs a header, we create a new object with a structure type that contains two fields, one for the header and one for the original allocation as shown below:

```
struct attribute((packed)) newTy {
  HeaderTy hd;
  Ty obj; // Ty is an original object's type
};
```

We insert a callsite to our hook function that decides if it is small or large-framed, updates metadata in the header, and also in the entry for large-framed objects. It then creates a flag and tag (offset or N value), and moves the pointer to the second field whose type is the actual allocated type by the target program. The hook returns a tagged pointer. The allocation of the original object is removed by FRAMER's pass, after the pass replaces all the pointers to the original object with the tagged pointer to the new object.

We instrument function epilogues to reset entries for large-framed non-static objects. Currently we instrument all the epilogues, but this instrumentation can be removed for better performance.

4.2.2 Statically-allocated objects (address-taken globals). Transformation on static/global objects is similar to handling stack objects. Creating a new global object with a header attached is straightforward, however, other parts of the implementation are more challenging.

For stack objects, FRAMER's pass replaces pointers to an original object with a tagged one (i.e. the return value of the hook). This cannot be applied to global objects, since the return value of a function is non-constant, whereas the original pointer may be an initializer of other static/global objects or an operand of `constant expression` (LLVM ConstExpr) [28]. Global variables' initializer and ConstExpr's operands must be constant, hence, the operations performed in a hook for stack objects should be done by a transformation pass for global objects.

In addition, while the tag should be generated at compile-time, the wrapper frame size is determined by their actual addresses in memory, that are known only at run-time. To implement a tagged pointer generated from run-time information at compile-time, FRAMER's transformation pass builds ConstExpr of (1) the wrapper frame size N (2) offset, (3) tag and flag selection depending on its wrapper frame size, (4) pointer arithmetic operation to move the pointer to the second field, and then finally (5) constructs a tagged pointer based on them. The original pointers are replaced with this constant tagged pointer. The concrete value of the tagged

Table 1: FRAMER inserts code, highlighted in gray, for creating a header-padded object, updating metadata and detecting memory corruption. Codes in line 2, 5, and 8 in the first column are transformed to codes in the second column.

	Original C	Instrumented C
1		`struct HeaderTy {unsigned size; unsigned type_id;};` `struct newTy{HeaderTy hd;int A[10];};`
2	`int A[10];`	`struct newTy new_A;`

3		`tagged = handle_alloc(&new_A, A_size);`
		`/* tagged = tag & &(new_A->A[0]),`
		`A_size = sizeof(int) * 10 */`

4	`int *p;`	`int *p;`
5	`p = A+idx;`	`p = tagged + idx;`
6		`check_inframe(tagged, p);`
7		`untagged_p = check_bounds(p, sizeof(int));`
8	`*p = val;`	`*untagged_p = val;`

pointer is then propagated at run-time, when the memory addresses for the base and bound are assigned.

FRAMER inserts at the entry of the program's main function a call to an initialisation function for each object. This function updates metadata in the header and, for large-framed objects, the address in the table entry, during program initialisation.

4.2.3 Heap objects. We interpose calls to `malloc`, `realloc`, and `calloc` at link time with wrapper functions in our binary libraries. The wrappers increase the user-defined size by the header size, call the wrapped function, and perform the required updates and adjustments similar to the hook for stack objects. We also interpose `free` with a wrapper to reset table entries for large-framed objects.

4.3 Memory Access

FRAMER's transformation pass inserts a call to our bounds checking function right before each `store` and `load`, such that each pointer is examined and its tag stripped-off before being dereferenced. The hook extracts the tag from a pointer, gets the header location, performs the check using metadata in the header, and then returns an untagged pointer after cleaning the tag. The transformation pass replaces a tagged pointer operand of `store`/`load` with an untagged one to avoid segmentation fault caused by dereferencing it.

Bounds checking and untagging are also performed on `memcpy`, `memmove` and `memset` in similar way. (Note that LLVM overrides the C lib functions to their intrinsic ones [29]). `memmove` and `memcpy` has two pointer operands, so we instrument each argument separately.

As for string functions, we interpose these at link time. Wrapper functions perform checks on their arguments, call `wrapped` functions with pointers cleared from tags, and then restore the tag for their return value.

4.4 Interoperability

FRAMER ensures *compatibility* between instrumented modules and regular pointer representation in precompiled non-instrumented libraries. We strip off tagged pointers before passing them to non-instrumented functions. FRAMER adds a header to objects for tracking, but this does not introduce incompatibility, since it does not change the internal memory layout of objects or pointers.

5 FRAMER APPLICATIONS

In this section we discuss how FRAMER can be used for building security applications. We explore mainly spatial safety, but we discuss additional case studies related to temporal safety.

5.1 Spatial Memory Safety

FRAMER can be used to track individual memory allocations, and store object bounds in the header associated with the object. These bounds can be used at runtime to check memory accesses. Unlike other object-tracking or relative location-based approaches, FRAMER can tackle legitimate pointers outside the object bounds without padding objects, or requiring metadata retrieval or bounds checking at pointer arithmetic operations.

In this section, we describe how FRAMER performs bounds checking at run-time.

5.1.1 Memory allocation. As described in Section 4.2, a header is prepended to memory objects (lines 1, 2 in Table 1). For spatial safety, this header must hold at least the raw object size, but can hold additional information such as a type id. This could be used for additional checks for sub-object bounds violations or type confusion. Its potential in type confusion checking is presented in Section 5.3, and we do not experiment with these in this work.

Once we get the header address from a tagged pointer, an object's base address is obtained by adding the header size to the header address. After a new object is allocated, a hook (`handle_alloc`) updates metadata, moves the pointer to (`new_A-¿A`), and then tags it (line 3). The pointer to the removed original object is replaced with a tagged one (`A` to `tagged` in line 5).

5.1.2 Pointer arithmetic. Going out-of-bounds at pointer arithmetic is not corrupting memory as long as the pointer is not dereferenced. However, skipping checks at pointer arithmetic can lose track of pointers' *intended referents*. Memory access to these pointer can be seen *valid* in many object bounds-based approaches. To keep track of intended referents, object-tracking approaches may have to check bounds at pointer arithmetic [22]. However, performing bounds checks only at pointer arithmetic may therefore cause *false positives*, where a pointer going out-of-bounds by pointer arithmetic is not dereferenced as follows:

```
int *p;
int *a = malloc(n * sizeof(int));
for (p = a; p ¡ &a[100]; p++) *p = 0;
```

On exiting the `for` loop, `p` goes out-of-bounds yet is not dereferenced – this is valid according to the C standard. [3] handles this by marking such pointers during pointer arithmetic and reporting errors only when dereferenced, and [22] pads an object by *off-by-one byte*.

Instead of padding, we include one *imaginary* off-by-one byte (or multiple bytes) when deciding the wrapper frame (see Section 3.2) on memory allocation. The fake padding then is within the wrapper frame, and pointers to this are still derived to the header, even when they alias another object by pointer arithmetic. The biggest advantage of fake padding is that it is allowed to be overlapped with neighboring objects and thus saves memory. The fake padding does not cause conflicting supplementary table N values across objects possibly overlapping the bytes.

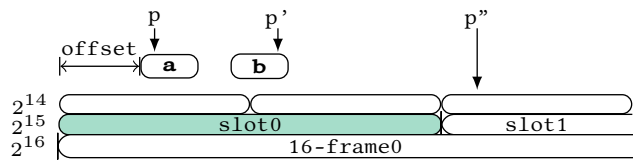


Figure 7: By pointer arithmetic, a pointer p goes out-of-bounds (p'), and also violates its intended referent (a to b). FRAMER still can keep track of its referent, since p' is *in-frame*. p" is *out-of-frame*, which we catch at pointer arithmetic.

FRAMER tolerates pointers to the padding at pointer arithmetic, and reports errors on attempts to access them. FRAMER detects those pointers being dereferenced, since bounds checking at memory access retrieves the raw size of the object. Currently FRAMER adds fake padding only in the tail of objects, but it could be also attached at the front to track pointers going under lower bounds, even though such pointer are banned by the C standard.

Beyond utilising fake padding, to make a stronger guarantee for near-zero false negatives, we could perform *in-frame checking* (currently not included for evaluation) at pointer arithmetic (line 6 in Table 1). We can derive the header address of an intended referent, as long as the pointer stays inside its wrapper frame (slot for small-framed), in any circumstance. In Fig. 7, consider a pointer (p), and its small-framed referent (a). Assuming p going out-of-bounds to p' by pointer arithmetic, p' even violates its intended referent, but p' is still within slot0. Hence, p' is derived to a's header by zeroing lower $log_2(slot\_size)$ (15) bits and adding offset. This applies the same for large-framed objects.

Hence, we could check only *out-of-frame* (p" in Fig. 7) by performing simple bit-wise operations (no metadata retrieval) checking if p and p' are in its wrapper frame (or slot for small-framed):

```
// p: the source pointer of pointer arithmetic
// p': the result of pointer arithmetic
// N: log2 wrapperframesize (or slotsize)
isinframe = (p'  p) & (0ULL ¡¡ N);
assert(isinframe == 0);
```

FRAMER may report false positives for programs not comforming to the C standard with out-of-frame pointers getting back in-frame by pointer arithmetic without being dereferenced while they are out-of-frame. This is very rare, and those uses will be usually optimised away by the compiler above optimisation level -O1. Normally the distance between an object's and its wrapper frame's bounds is large. We can also increase the wrapper frame size for all objects to enlarge this distance.

There is another rare case of false positives (we did not encounter them), where library code uses a tagged pointer it reads from memory, where the instrumentation did not have a chance to clear the tag (the pointer was not passed as a function argument). This can be handled with hardware support or, with a performance overhead, by a segmentation fault handler.

5.1.3 Memory access. As mentioned in Section 4.3, we instrument memory access by replacing pointer operands with a return of our hook, so that the pointers are verified and tag-stripped, before being dereferenced (line 7,8 in Table 1).

check_bounds first reads a tagged pointer's flag telling if the object is small or large-framed. As we described in Section 3.3.1 and 3.3.2, we derive the header address from either an offset or an entry, and then get an object's size from the header and its base address as follows:

```
objbase = headeraddr + sizeof(HeaderTy);
objsize = ((HeaderTy *)headeraddr)-¿size;
```

We then check both under/overflows ((1) and (2) below, respectively). Detection of underflows is essential for FRAMER to prevent overwrites to the header.

```
assert(untaggedp ¿= objbase);
      // (1)
assert(untaggedp + sizeof(T) - 1 ¡= upperbound)
    ); // (2)
// Where T is the type to be accessed
```

The assertion (2) aims to catch overflows and memory corruption caused by access after unsafe typecast such as the following example:

```
char *p = malloc(10);
int *q = p + 8;
*q = 10; // Memory corruption
```

In a similar fashion, we instrument memcpy, memmove, memset, and string functions (strcpy, strncmp, strncpy, memcmp, memchr and strncat). Handling individual function depends on how each function works. For instance, strcpy copies a string src up to null-terminated byte, and src's length may not be equal to the array size holding it. As long as the destination array is big enough to hold src, it is safe, even if the source array is bigger than the destination array. Hence, we check if the destination size is not smaller than strlen(src), returning the length up to the null byte as follows:

```
assert(destarraysize ¡= strlen(src));
```

On the other hand, strncpy copies a string up to user-specified n bytes, so we check both sizes of destination and source arrays are bigger than n. Metadata for both arrays are retrieved for bounds checking unlike handling strcpy.

5.2 Temporal Memory Safety

Although our primary focus in this paper is spatial safety, FRAMER can also detect some forms of temporal memory errors [2, 11, 33, 40] that we now discuss briefly.

Each large-framed object is mapped to an entry in a division array in the supplementary table, and the entry is mapped to at most one large-framed object for each N. We make sure an entry is set to zero whenever a corresponding object is released. This way, we can detect an attempt to free an already deallocated object (i.e. a *double free*), by checking if the entry is zero. Access to a deallocated object (i.e. *use-after-free*) is detected in the same way during metadata retrieval for a large-framed object. Note that this cannot detect invalid *temporal* intended referents, i.e., an object is released, a new object mapped to the same entry is allocated, and then a pointer attempts to access the first object.

Detection of dangling pointers for small-framed objects is out-of-scope for this case-study.

5.3 Type Cast Checking

FRAMER can be used for other applications such as type safety, garbage collection and etc. We did not implement in the aspect in this paper, but we briefly introduce how to utilise FRAMER as the base of type safety enforcement as an application in this section.

The majority of type casts in C/C++ programs are either *upcasts* (conversion from a descendant type to its ancestor type) or *downcasts* (in the opposite direction). Upcasts are considered safe, and this can be verified at compile time, since if a source type of upcasts is a descendant type, then the type of the allocated object at runtime is also a descendant type.

In contrast, the target type of downcasts may mismatch the runtime type (RTT). If an allocated object's type is a descendant type of the target type at downcast, access to an object after downcasts may cause boundary overruns including internal overflows. This is a vulnerability commonly known as *type confusion* [16, 21**? ?**]. The RTT is usually unknown statically due to inter-procedural data flows, so downcasts require run-time checking to prevent this type confusion.

RTT verification is more challenging than upcast checking at compiler-time, since it requires *pointer-to-type* mapping. We need to track individual objects (or pointers) and store per-object (pointer) type information in the database. In addition, RTT checking requires mappings of unique offsets to fields corresponding to types of sub-objects. FRAMER could be the basis of metadata storage (mapping a pointer to per-object type) with supplementary type descriptors. FRAMER's header can hold corresponding per-object type layout information (i.e. a list of types at each offset in the object type) or its type ID for the object, and all type layout information and type-compatibility relations can be stored in the type descriptors (implementations can vary). FRAMER's current implementation as an LTO pass makes it easier to collect all used types of the whole program.

Downcasts may be critical for approaches using embedded metadata (e.g. fat pointers or tagged pointers), since memory writes after unsafe type casts on program's user data can pollute metadata in a neighboring object's header. Prevention of metadata corruption is easier with FRAMER than with fat pointers. We can detect memory overwrites to another object's header caused by downcasts by simply keeping track of structure-typed objects and using our bounds checking. Unlike fat pointers, we do not need to check internal overflows by unsafe downcasts to protect metadata, since metadata is placed outside an object.

6 OPTIMISATIONS

We applied both our customised and LLVM built-in optimisations. This section describes our own optimisations. Suggestion of further optimisations is provided later in Section 8.3.

Implementation Considerations. As described in Section 4.2.2, we replaced all occurrences of an original pointer to a global object with a tagged one in constant expression (LLVM ConstExpr). Unfortunately, we experienced runtime hotspots due to the propagation of a constant (a global variable's address) to every large ConstExpr. To work around this issue we created a *helper* global variable for each global object; assigned the result of the constant propagation to the corresponding helper variable during program initialisation; and then replaced uses of an original pointer with `load` of the helper variable. This way, runtime overheads are reduced, for instance, benchmark `anagram`'s overhead decreased from 14 to 1.7 seconds.

Non-array Objects. We do not track non-array objects that are not involved with pointer arithmetic, e.g., int-typed objects. It is redundant to perform bounds checking or untagging for pointers to them. We filter out simple cases, easily recognised, from being checked. In the general case, it is not trivial to determine if a pointer is untagged at compile time, since back-tracing the assignment for the pointer requires whole-program static analysis.

Safe Pointer Arithmetic. Instead of full bounds checks, we only strip off tags for pointers involved in pointer arithmetic and statically proven in-bound for simple cases. For pointers where the bounds can be determined statically, we checks if the index is smaller than the number of elements.

In some SPEC benchmarks, there are statically proven out-of-bound accesses, but we do not report memory errors since they may be unreachable. We inserted a termination instruction for this case so that it can report errors at runtime, when the execution reaches the point.

Hoist Run-time Checks Outside Loops. Loop-invariant expressions can be hoisted out of loops, thus improving run-time performance by executing the expression only once rather than at each iteration. We modified SAFECode's [12, 41] loop optimisation passes. We apply hoisting checks to monotonic loops, and pull loop invariants that do not change throughout the loop, and scalars to the pre-header of each loop. This pass works on each loop and if there are inner loops, it handles them first. While iterating our run-time checks inside each loop including inner loops, we determine if the pointer is hoistable. If a pointer is hoistable, we place its scalar evolution expression along with its run-time checks outside the loop, and delete the checks inside loop.

Inlining Function Calls in the Loop. Inlining functions can improve performance, however it can bring more performance degradation due to the bigger size of the code (runtime checks are called basically at every memory access). Currently, we only inline bounds checks that are inside loops to minimise code size.

7 EVALUATION

We measured the performance of FRAMER on C benchmarks from Olden [7], Ptrdist [4], and SPEC CPU 2006 [18]. For each benchmark we measured four binary versions: uninstrumented, only store-checked and full (both load and store checking enabled) on FRAMER, and ASan – one of the most widely used sanitizers. We disabled ASan's memory leak detection at run-time and halt-on-error to measure overheads in the same setting as FRAMER. Binaries were compiled with the regular `LLVM-clang` version 4.0 at optimisation level `-O2`. Measurements were taken on an Intel® Xeon® E5-2687W v3 CPU with 132 GB of RAM. Results were gathered

Table 2: Summary averages over all benchmarks (first three columns normalised)

	Memory footprint	Runtime (cycles)	Dynamic instructions	IPC	Load density	D-cache MPKI	Branch density	B-cache MPKI
Baseline	1.00	1.00	1.00	1.70	0.28	24.85	0.19	2.85
Store-only	1.22	1.70	2.24	2.17	0.20	12.27	0.15	1.34
Full check	1.23	3.23	5.25	2.54	0.14	5.28	0.17	0.86

using `perf`. Table 2 summarises the average of metrics of the baseline and the two instrumented tests.

In this text, cache and branch misses refer to L1 D-cache misses and branch prediction misses both per 1000 instructions (MPKI), respectively.

7.1 Memory Overhead

Our metadata header was a generous 16 bytes per object. The large-frame array had 48 elements for each 16-frame (division) in use where the element size was 8 bytes to hold full address of the header. The header size and the number of elements of each division array can be reduced. Currently we mandate 16 alignment for compatibility with the `llvm.memset` intrinsic function that sometimes assumes this alignment. Despite inflation of space using larger than needed headers and division array entries and some changes of alignment, we see FRAMER's space overheads are very low at 1.22 and 1.23 as shown in Fig. 9. These measurements reflect code inflation for instrumenting both loads and stores.

The memory overheads of FRAMER are low and stable compared to other approaches [32, 39]. ASan's average normalised overheads are 8.84 for the same working set in our experiments, and the highest overhead is 4766% for `hmmer`. The average memory overhead of FRAMER is 22% $\sim$ 23% for both store-only and full checking, and only two tests, `perlbench.2` (84%) and `yacr2` (116%) recorded comparably higher growth than other tests. The two tests produce many small-sized objects, for example, `perlbench` allocates many 1-byte-sized heap objects. Currently FRAMER instruments every heap object, so attaching a 16-byte-sized header to all the 1-byte-sized objects made the increase higher. FRAMER's overheads for those benchmarks are still much lower than ASan's: 2808% for `perlbench.2` and 714% for `yarc2`.

7.2 Slowdown

Fig. 8 reports the slowdown per benchmark (relative number of additional cycles). The average is 70% for store-only and 223% for full checking. For full-checking, `anagram` (410%) and `ks` (452%) stand out for high overheads despite its smaller program size, mostly due to heavy recursion and excessive allocations causing big growth in executed instructions (674% for `anagram`, 812% for `ks`) as shown in Fig. 12, but decreases in cache misses are moderate (76% for `anagram`, 81% for `ks`) compared to average (decreased by 63%). On contrast, `mcf` recorded the highest instruction overheads (1097%), but cache (91%) and branch misses (92%) are dropped the biggest among all the tests, so run-time overhead did not grow in proportion to increased instruction count. `perlbench` and `bzip` sets' overheads are high in both FRAMER and ASan. Both tests produce many objects, and especially `bzip` recorded much higher growth in executed instructions than `perlbench` and others.

Performance was impacted far less than would naively be expected from the additional dynamic instruction count (metric columns 2 and 3 in Table 2). The rise in IPC (column 4) is quite considerable on average, although the figure varies greatly by benchmark. The original IPC ranged from 0.22 to 3.20 but after instrumentation there was half as much variation.

Our slowdown is mainly due to increased dynamic instructions to calculate metadata location. We measured runtime overheads for metadata management/retrieval (excluding bounds checking) of benchmarks with the highest runtime overheads by forcing or preventing inline our hooks. As shown in Fig. 10, the fluctuations in proportion is negligible. Benchmarks with low runtime overheads showed a similar pattern.

Slowdown is dominated by Calculation (69.66%) – ALU operations to (1) derive the header address at memory access and (2) generate a tag at memory allocation. Hardware acceleration in a future ISA would largely resolve this overhead. We isolated tag-cleaning from Calculation to show the cost of using tagged pointers without hardware support. Its cost (6.07%) would be removed on current ARM that ignores top spare bits. The cost of generating tags was negligible, since it is performed only at allocation.

The remaining 3 components cannot be resolved with simple ISA changes. Branch checking for tagged/untagged and small/large-framed contributes 10.19% of the total overheads of metadata management. Current FRAMER encoding avoided any restriction on object alignment, however, we are open to manipulate memory manager to remove large-framed objects for the future design. Accessheader and Accessentry represent ratios of overheads to access a header and entry, once their addresses are calculated from tagged pointers. Accessing a header takes more time than accessing an entry, since it is performed on both types of objects. Excluding the overhead of arithmetic operations, the cost is around 25% of that of metadata management and retrieval.

The remaining part of the total runtime overhead that is not included in the measurement shown in Fig. 10 is bounds checking performing arithmetic operations with loaded metadata, which can be resolved by ISA.

7.3 Data Cache Misses

One of the goals of FRAMER is to allow flexible relationships between object and header locality to minimise additional cache misses from metadata access. We do not analyse L1 instruction cache miss rate since this generally has negligible performance effect on modern processors, despite our slightly inflated code. To explain the measured increase in IPC we analyse L1 D-cache misses

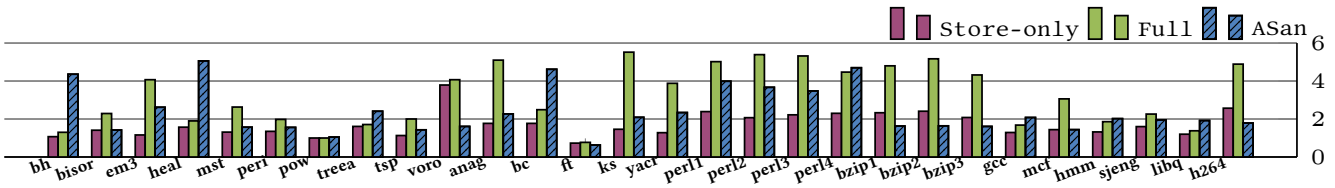


Figure 8: Normalised runtime overheads

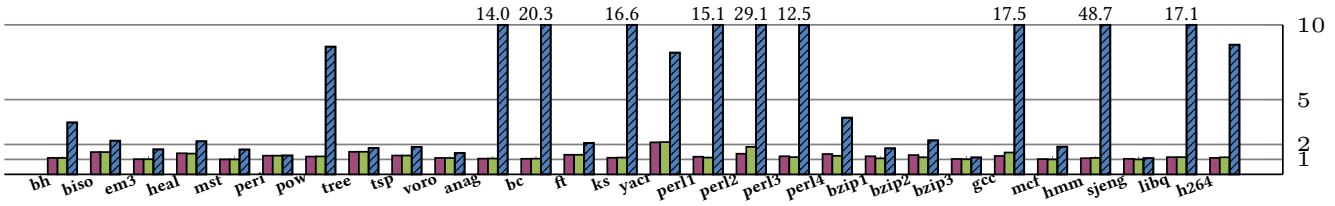


Figure 9: Normalised maximum resident set size

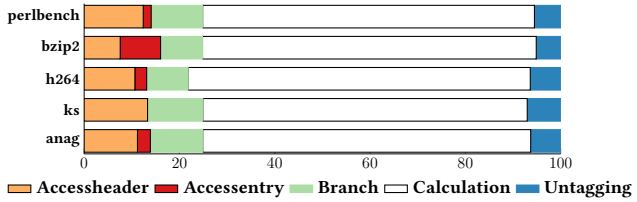


Figure 10: Runtime overheads for metadata management and retrieval (the overhead for bounds checking excluded)

MPKI (cache misses) and branch prediction misses MPKI. The baseline D-cache miss rate was 2.48% (Table 2) but this improves with FRAMER enabled owing to repeated access to the same cache data.

In Fig. 11, we normalise cache misses to the uninstrumented figure. The average normalised cache misses is 0.66 and 0.38 for store-only and full-checking, respectively. The miss rate is reduced since the additional operations we add have high cache affinity which dilutes the underlying miss rate of the application.

While ASan showed increase for four tests. ASan's normalised misses on average is 0.73, which is higher than FRAMER's 0.38. ASan's highest overhead is 197% for `bc`, and two tests reached increase more than by 100%. On FRAMER, `power`'s overhead by 48% is mainly caused by the very low increase in instruction executed in producing MPKI. The rest of benchmarks' misses decreased, and normalised misses in full-checking mode were below 0.5 for 21 tests among 28 working set, whereas only 13 tests' on ASan were lower than 0.5. The overall cache miss rate showed FRAMER is cache-efficient and stable.

Cache misses (MPKI) may appear decreased with bloated instruction counts, so we also present the increase in total numbers of cache misses. Fig. 13 shows the normalised counts of cache misses for big-sized programs in SPEC. The averages of shown tests for FRAMER (Full) and ASan are 1.24 and 2.40, and the averages for the whole set are 1.40 and 2.31, respectively. This shows the increase in cache miss count to access metadata in FRAMER is minimal. On FRAMER, the increase rate of all the tests except one (277% for `voronoi`) are below 100%. On ASan, the increases for 7 tests are above 100%, and `bc`'s increase rate is 1160%.

7.4 Instructions Executed

Fig. 12 reports normalised overheads per benchmark. FRAMER increases dynamic instruction count by 124% for store-only, and 425% for full checking. This increase is the main contributor to slowdown. Dynamic instruction penalty arises from setting up and using tagged pointers. The major source of the growth is arithmetic operations. As shown in Fig. 10, 75% of runtime overhead of metadata management/retrieval is dominated by calculation of (1) the header address at memory access and (2) the tag at allocation. This cost can be resolved with hardware acceleration with ISA.

The penalty of utilising top bits is *over-instrumentation* – unless individual memory access is proven tag-free statically, we have to instrument it (i.e. tag-cleaning) to avoid segmentation fault in all major architectures, requiring the top bits to be zero (or special pointer authentication code in ARM8). This results in stripping the tag field for untagged pointers.

The average overhead for ASan is 226%, which is lower than FRAMER. The average excluding the highest test (1336% for `bh`) is 184%, while FRAMER's average excluding the highest (1098% for `mcf`) is 400%. The difference in slowdown on average (FRAMER: 213%, ASan: 139%) was not big as the difference of instruction executed due to FRAMER's cache efficiency. ASan consumes fewer dynamic instructions, since shadow space-only metadata storage helps simpler derivation of metadata location, taking advantage of re-alignment of objects, as trade-off of space and high locality.

Future implementations can optimise the case where conservative analysis reveals the tag never needs to be added. More discussion on optimisation is described in Section 8.3.

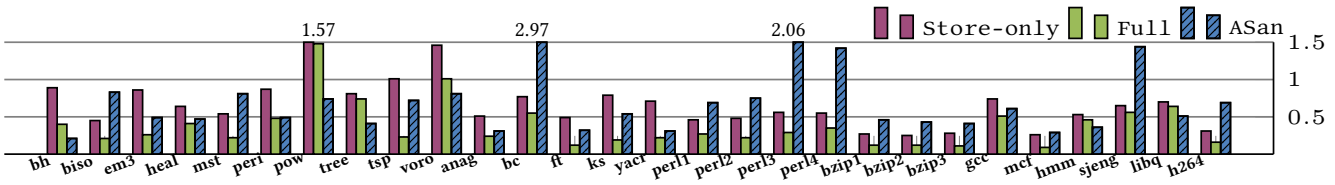


Figure 11: Normalised L1 D-cache load misses per 1000 instructions (MPKI)

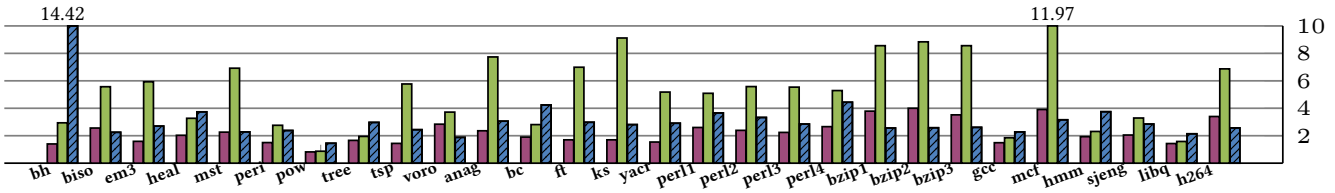


Figure 12: Normalised dynamic instruction count

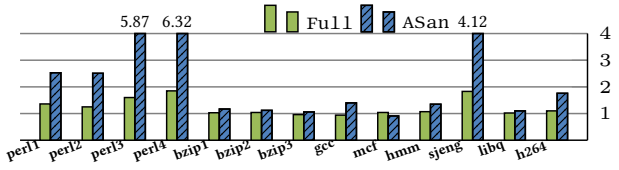


Figure 13: Normalised L1 D-cache load miss count

7.5 Branch Misses

Additional conditional branches arise in FRAMER from checking whether small or large frame is used and in the pointer validity checks themselves. Many approaches using shadow space are relieved from these branches at metadata retrieval.

As shown in Table 2 col 7, the dynamic branch density decreases slightly under FRAMER instrumentation, but the branch mis-prediction rate greatly decreases (col 8). The averages of normalised branch misses for store-only and full-checking are 0.62 and 0.42, respectively. This shows the additional branches achieve highly accurate branch prediction and that branch predictors are not being overloaded. Of the new branches added, the ones checking small/large frame size are completely statically predictable owing to the checking code instances being associated with a given object. And the ones checking pointer validity also predict perfectly since no out-of-bounds errors are detected.

8 DISCUSSION

8.1 Comparison with Other Approaches

8.1.1 Shadow space-based approaches. Shadow space-based approaches reduce slowdown by lowering executed instructions. Trade-off of data memory is tolerable in most systems during development. For practical deployment, however, their slowdown is still high and memory footprint is critical in some systems, e.g. ARM running in an embedded system or I/O-heavy server-side loads. In using shadow space, it is inevitable to pad and re-align objects to avoid conflicts in entries [3, 25, 39]. ASan pads each object for wider detection coverage and more padding for alignment, which burdens space, whereas FRAMER's fake padding and wrapper frames do not consume any space. Furthermore, their higher cache misses to access metadata in remote memory region (including ASan's resetting entries at deallocation), making its runtime overheads unpredictable.

In comparison with ASan, FRAMER showed better efficiency both in memory footprint (FRAMER: 23%, ASan: 784%) and cache miss counts overhead (FRAMER: 40%, ASan: 131%). ASan showed lower increase in runtime overheads (FRAMER: 223%, ASan: 139%), however, 75% of FRAMER's overhead of metadata management and retrieval is consumed for calculation, that can be largely resolved with new ISA. The rest of overhead comes from bounds checking using loaded metadata, that can be also implemented as ISA.

8.1.2 SGXBounds. SGXBounds spares 32 bits for a tag among 64 bits, while FRAMER tags only upper spare 16 bits. SGXBounds's retrieving an upper bound first, not the base like FRAMER, may save some overheads if we perform overflow-only checking. However, using a footer makes systems slightly more vulnerable to metadata pollution without complete memory safety. For both over/under underflow checking, we do not consider our derivation of the base, not the upper bound, as a weakness. In addition, frame encoding can be easily integrated to SGXBounds' design.

8.1.3 MPX. In principle, FRAMER could utilize MPX extensions for performance when used for spatial safety. We showed FRAMER is more cache-friendly, but it could be made even faster if a single instruction implemented the complete tag decode operation, splitting apart the tagged pointer into an untagged object pointer and separate header pointer in another register. This would be a fairly simple, register-to-register instruction, operating on general purpose registers. Since this has not used the D-cache, an enhancement

would be to compare the pointer against a bounds limit at hard-coded offset loaded from the header, but the best design requires further study.

8.2 Hardware Implementation of FRAMER

We believe FRAMER's encoding is at its best when it is implemented as instruction set extensions. As mentioned in 8.1.3, the increase in the number of executed instructions for calculation, the main contributor to slowdown of FRAMER, can be resolved with new instructions. Tag-cleaning can be supported by hardware [27]. Moreover, generating a tag and deriving a metadata address can be implemented as a single operation, respectively.

8.3 Additional Optimisations

8.3.1 Utilising More Spare Bits. Currently, we mandate 16-alignment due to llvm.memset intrinsic function. On this alignment, we have spare 4 bits at the end of offset for small-framed and another 4 bits in the pointer. (We already have spare bits for large-framed ones.) Using the bits, we can perform bounds checking only at pointer arithmetic and mark out-of-bounds pointers, so that we can report errors when they are dereferenced. This way, we expect to remove duplicated runtime checks, since the pointer may be used for memory access multiple times. Above this, we can utilise them to encode more information for better performance.

8.3.2 Compiler Optimisation. Redundant runtime checks can be eliminated using *dominator trees*. SoftBound [32] reported that their simple dominator-based redundant check elimination improved performance by 13% and claimed more advanced elimination [6, 50] can reduce more overheads.

The penalty of using tagged pointers is that unless individual memory access is proven safe at static time, we may have to *over-instrument* memory access to avoid segmentations fault. Some approaches can save expensive runtime checks to reduce performance degradation, bearing false negatives, but it is difficult in approaches using tagged pointer. We did not run dedicated pointer-analysis for this version but it can remove over-instrumentation. Loop optimisation did not show big impact on reducing overheads, even for some SPEC benchmarks whose number of hoisted run-time checks reached hundreds at static time. Our naive optimisation skipping untagging improved performance more than state-of-the-art loop hoist pass. *Static points-to analysis* [43, 44], as long as it does not assume the absence of memory errors, potentially enables many tags and bounds checks to be removed at compile time.

9 CONCLUSION

We presented FRAMER, a per-object capability system utilising the currently unused significant bits of pointers to store a tag. A key insight is that this tag can be bifurcated using a flag bit so that the overwhelmingly common case of *small-framed* objects can be dealt with efficiently in terms of both time and space. This ultimately benefits the performance of exceptional *large-framed* objects too, because the design can special-case them as well.

We evaluated FRAMER with a case study on spatial memory safety in C programs. However, we believe its capability design could benefit the performance of other programming language security mechanisms as well. Compared to existing approaches, our *frame*-based offset encoding is more flexibile both in metadata association and memory management, while still offering a fairly simple calculation to map from arbitrary pointers to metadata locations. In addition, its intrinsic memory and cache-efficiency make it potentially attractive for direct hardware support.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, as well as Elias Athanasopoulos, Stephen Kell, Guy Lemieux, Min Hong Yun, Jonathan Woodruff, and Matt Staats for their valuable comments on the draft, and Christos Rikoudis for his helpful tips for troubleshooting during experiments. This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. 2017M3C4A7083676).

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proc. of ACM CCS*. 340–353.

[2] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1929820.1929836

[3] 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 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 51–66. http://dl.acm.org/citation.cfm?id=1855768.1855772

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

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/178243.178446

[6] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 321–333. https://doi.org/10.1145/349299.349342

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

[8] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC '06)*. IEEE Computer Society, Washington, DC, USA, 749–754. https://doi.org/10.1109/ISCC.2006.158

[9] Jack B. Dennis and Earl C. Van Horn. 1965. Programming Semantics for Multiprogrammed Computations.

[10] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. *SIGPLAN Not.* 43, 3 (March 2008), 103–114. https://doi.org/10.1145/1353536.1346295

[11] D. Dhurjati and V. Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *International Conference on Dependable Systems and Networks (DSN'06)*. 269–280. https://doi.org/10.1109/DSN.2006.31

[12] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 144–157. https://doi.org/10.1145/1133981.1133999

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

[14] Robert Gawlik and Thorsten Holz. 2014. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proc. of ACSAC*. 396–405.

[15] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out Of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*. 575–589.

[16] István Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion

Detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 517–528. https://doi.org/10.1145/2976749.2978405

[17] István Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. 2016. METAlloc: efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security, EUROSEC 2016, London, UK, April 18-21, 2016*. 5:1–5:6. https://doi.org/10.1145/2905760.2905766

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

[19] Intel Corporation. 2013. Introduction to Intel® memory protection extensions. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[20] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proc. of NDSS*.

[21] Yuseok Jeon, Priyam Biswas, Scott A. Carr, Byoungyoung Lee, and Mathias Payer. 2017. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2373–2387. https://doi.org/10.1145/3133956.3134062

[22] Richard Jones, , Richard W M Jones, and Paul H J Kelly. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. , pages 13–26 pages.

[23] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks Without the Checks. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/3190508.3190553

[24] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 205–221. https://doi.org/10.1145/3064176.3064192

[25] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, Gingko Bioworks, and Andre Dehon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. CCS.

[26] Arm Limited. 2018. ARM A64 Instruction Set Architecture. https://static.docs.arm.com/ddi0596/a/DDI_0596_ARM_a64_instruction_set_architecture.pdf.

[27] Arm Limited. 2018. Armv8.5-A. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a.

[28] llvm team. 2003. LLVM Constant Expression. https://llvm.org/docs/LangRef.html#constant-expressions.

[29] llvm team. 2003. Standard C Library Intrinsics. https://llvm.org/docs/LangRef.html#standard-c-library-intrinsics.

[30] llvm team. 2003. The LLVM gold plugin. https://llvm.org/docs/GoldPlugin.html.

[31] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 190–208. https://doi.org/10.4230/LIPIcs.SNAPL.2015.190

[32] 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)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[33] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. *SIGPLAN Not.* 45, 8, 31–40. https://doi.org/10.1145/1837855.1806657

[34] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[35] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 577–587. https://doi.org/10.1145/2594291.2594295

[36] 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

[37] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proc. of NDSS*.

[38] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 135–148. https://doi.org/10.1109/MICRO.2006.29

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[40] Matthew S. Simpson and Rajeev K. Barua. 2013. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Softw. Pract. Exper.* 43, 1 (Jan. 2013), 93–128. https://doi.org/10.1002/spe.2105

[41] Stacy Simpson. 2014. SAFECode Whitepaper: Fundamental Practices for Secure Software Development 2nd Edition.. In *ISSE*, Helmut Reimer, Norbert Pohlmann, and Wolfgang Schneider (Eds.). Springer, 1–32. http://dblp.uni-trier.de/db/conf/isse/isse2014.html#Simpson14

[42] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*.

[43] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

[44] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

[45] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proc. of USENIX SEC*. 941–955.

[46] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proc. of ACM CCS*. 927–940.

[47] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough `call`: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proc. of IEEE S&P*. 934–953.

[48] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

[49] Jonathan Woodruff, Robert N.M. 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 *ISCA '14: Proceeding of the 41st annual international symposium on Computer architecture*. IEEE Press, Piscataway, NJ, USA, 457–468. https://doi.org/10.1145/2678373.2665740

[50] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2007. Array Bounds Check Elimination for the Java HotSpot™ Client Compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ '07)*. ACM, New York, NY, USA, 125–133. https://doi.org/10.1145/1294325.1294343

[51] Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/1029894.1029913

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

[53] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *Proc. of NDSS*.

[54] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. 2013. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proc. of IEEE S&P*. 559–573.

[55] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX SEC*. 337–352.

A PROOFS

A.1 Proof 1

Given an object o and its wrapper frame f, let's assume there exists a smaller frame x that has o inside. Since o resides in both f and x, we can conclude that x is a subframe of f. According to the assumption, the base address of o ($base_o$) is within the range of x, hence, we get $base_x \leq base_o$. Here, f is o's wrapper frame, so $base_o$ is placed in f's lower subframe. x is a subframe of f, hence x must be f's lower subframe. This is resolved to contradiction between the assumption (x has o inside) and the definition of wrapper function (o's upper bound in the upper subframe). Hence, we can conclude that there is no smaller frame than o's wrapper frame; this is actually the unique wrapper frame, and it can be used as a reference point.

A.2 Proof 2

We prove that for each N, there exists at most one N-object mapped to each entry of a division array, and show N identifies an object mapped to the same division array. To prove this, we assume there exist two distinctive objects, x and y; both are N-objects ($N \geq 16$) mapped to the same division array. Since x and y are N-objects, their wrapper frame (f_x and f_y) is 2^N-sized by definition. The division is the only one that f_x and f_y are mapped to as shown previously, so f_x and f_y have the same base address as the division. In addition, both frames have the same size, so they are identical. Both base addresses of x and y (b_x, b_y) must be in the lower $(N-1)$-subframe of f_x (or f_y), and end addresses must be in the other sub-frame. From this, b_x and b_y must be smaller than e_x and e_y. However, the objects are distinct, so $b_x < e_x < b_y < e_y$ or vice versa must hold. The assumption leads to a contraction. We conclude that for each N, there is a unique N-object mapped to one division array.