Backward-Compatible Constant-Time Exception-Protected Memory

Pradeep Varma
IBM India Research Laboratory
4, Block C, Institutional Area
Vasant Kunj, New Delhi 110070
+91-11-41292140, +91-11-26138889(FAX)
pvarma@in.ibm.com

Rudrapatna K. Shyamasundar
Faculty of Technology and Computer Science, Tata Institute of Fundamental Research, Mumbai 400005, +91-22-22804777
shyam@tifr.res.in

Harshit J. Shah
School of Technology and Computer Science, Tata Institute of Fundamental Research, Mumbai 400005
harshit@tcs.tifr.res.in

ABSTRACT

We present a novel, table-free technique for detecting all temporal and spatial memory access errors (e.g. dangling pointers, out-of-bounds check, etc.) in programs supporting general pointers. Our approach is the first technique to provide such error checking using only constant-time operations. The scheme relies on fat pointers, whose size is contained within standard scalar sizes (up to two words) so that atomic hardware support for operations upon the pointers is obtained along with meaningful casts in-between pointers and other scalars. Optimized compilation of code becomes possible since the scalarized-for-free encoded pointers get register allocated and manipulated. Backward compatibility is enabled by the scalar pointer sizes, with novel automatic support provided for encoding and decoding of fat pointers in place for interaction with unprotected code (e.g. library binaries). Implementation and benchmarks of the technique over several applications of the memory-intensive Olden suite indicate that the average time overhead of our method is about half the time cost of an unprotected application's execution (< 55%). This performance is over twice faster than the nearest prior work.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *Data types and structures, dynamic storage management;* D.2.5 [**Software Engineering**]: Testing and Debugging – *Error handling and recovery, debugging aids;* D.3.4 [**Programming Languages**]: Processors – *Run-time environments*

General Terms

Algorithms, Design, Languages, Performance, Reliability

Keywords

Memory safety, backward compatibility, object version, scalar fat pointer, spatial access error, temporal access error

ESEC/FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.

1. INTRODUCTION

Memory safety in the context of C/C++ became a concern a decade or so after the advent of the languages [10]. Austin et al. [1] described a *memory access error* as a dereference outside the bounds of the referent, either address-wise or time-wise. The former comprises a *spatial access error* e.g. array out of bounds access error, and the latter comprises a *temporal access error* e.g. dereferencing a pointer after the object has been freed. Austin et al. provided the first system to detect such errors relatively precisely (viz. temporal access errors, whose treatment earlier had been limited). However, the work had limited efficiency (temporal error checks had a hash-table implementation with worst-case linear costs; for large fat pointer structures, register allocation was compromised with accompanying performance degradation; execution-time overheads were benchmarked above 300%). The fat pointers also compromised backward compatibility [22]. Significant work has transpired since [1] on these error classes because of the very hard to trace and fix attributes of these errors [2, 5, 6-9, 11-19, 22]. The insight of Austin et al. into temporal access errors, namely that object lifetimes can be caught as a pointer attribute, a *capability*, has led to several works – Electric Fence, PageHeap, its follow-ons [8], and [22].

We continue in this tradition, making one key, novel departure from these earlier works. There is no capability store or table or page table in our work that is required to be looked up each time an object is accessed. Our notion of a capability is an object *version* that is stored with the object itself and thus is available in cache with the object for lookup within constant time. In effect, an object for us is the C standard's definition [4], namely, a storage area whose contents may be interpreted as a value, and a version is an instantiation or lifetime of the storage area.

With this, the overheads for temporal access error checking in our work can asymptotically be guaranteed to be within constant time. Furthermore, since each object has a version field dedicated to it, the space of capabilities in our work is partitioned at the granularity of individual objects and is not shared across all objects as in [1, 22] and is more efficient than a capability as a virtual page notion of Electric Fence, PageHeap and [8]. This feature lets our versions be represented as a bitfield within the word that effectively contains the base address of the referent (as an offset into a pre-allocated protected heap), which means that we save one word for capabilities in comparison to the encoded fat pointers of [1] without

compromising on the size of the capability space.[1] Since versions are tied to objects, the object or storage space is dedicated to use solely by re-allocations of the same size (unless garbage collector (gc) intervenes). This fixedness of objects is put to further use by saving the referent's size with the object itself (like version), saving another word from the pointer metadata.

These savings that we make on our pointer metadata are crucial in bringing our encoded pointers down to standard scalar sizes of one or two words in contrast to the 4-plus words size of [1] and a similar price of [22]. Standard scalar sizes means that our encoded pointers assist backward compatibility, avail of standard hardware support for atomic reads and writes, and can be meaningfully cast to/from other scalars, and achieve higher optimization via register allocation and manipulation.

Like [22] our work detects memory access errors at the level of memory blocks. A memory block comprises the least unit of memory allocation such as a global or local variable, or the memory returned by a single invocation of malloc. Our work detects all memory errors at this level, except for uninitialized data reads, where it does more (than [1, 22]), by flagging all uninitialized data reads and not just uninitialized pointer reads using a Purify-like approach. The coverage of uninitialized data reads in this manner is complete for small objects, and is approximate for large objects.

By detecting memory access errors at the level of memory blocks, our work targets the general pointer arithmetic model supported by C [4], with dereferences disallowed only when they cross allocation bounds and not while they remain within. So for instance, a safe memcpy() can be written that takes an element pointer of a struct and copies up or down without exception so long as it remains within the allocated memory for the struct. Arithmetic can cause a pointer to cross allocation boundaries arbitrarily, only dereferences have to be within the allocated memory as in Ruwase et al. [18] and not as in Jones et al. [12].

Fat pointer approaches like [1] have suffered from backward compatibility problems because fat pointers change structure layouts. C programs often assume that the size of a pointer is the same as that of a long integer in structure layouts. A union or a cast from a pointer to an integer may make similar assumptions. These assumptions break when large fat pointers are used in place of normal pointers as in [1]. Library binaries, compiled for non-fat pointers index structures using offsets that mismatch the fields of structures containing fat pointers. For these reasons, [22] diverged from [1] in storing pointer metadata separately from the pointers themselves. While this improves backward compatibility somewhat, [22] is still hobbled by having to pass meta-data parameters to functions separately from the pointer parameters, forcing interface changes with functions for both parameters passed in and results returned back (Section 2.3.4, [22]). There is also no support in [22] for generating the metadata associated with unknown pointers returned by library functions.

Our work provides much better support for backward compatibility than [1] or [22] using scalar-sized fat pointers. There are two incarnations of our general pointer layouts – the general-heap layout, and a reduced-heap layout. The general layout uses a two-word scalar representation of the general pointer and the reduced-heap layout uses a one-word scalar representation. Backward compatibility offered by reduced-heap layouts is ideal – the encoded general pointer has the same scalar size as an un-encoded pointer (one word). Similarly, the backward compatibility offered by the simpler version of our pointers (Section 3) is ideal – it provides full heap sizes and 1-word encoded pointers. These pointers can be used with pre-compiled libraries with very effective backward compatibility. The general pointer layout (2 word scalar) would also offer similar compatibility if it were possible to obtain vendor libraries in which pointer sizes are double-word scalars[2]. Once encoded pointers and un-encoded pointers of the same scalar size have been obtained, backward compatibility reduces to the ability to provide un-encoded versions of the pointers to a library via arguments and encoded versions of the same to application code when the library returns results. For this, novel, automatic support for encoding and decoding of pointers is provided. So a library can continue with processing un-encoded pointers only while the application deals with encoded pointers alone and the interface uses the automatic support to transform pointers in place between the application and the libraries. Similarly, unprotected code manipulating pointers as integers can be provided un-encoded pointers at the time of the cast to integer and un-encoded pointers obtained from a cast from integers can be converted into encoded pointers using this support.

All capability-based systems, e.g. our reduced-heap system have a problem that they can run out of capability space (i.e. version space for us). This is because the capability fields have a fixed size and hence the number of capabilities they represent is fixed while a long-running program can engender an unbounded number of object lifetimes. Except for [8], which approaches this issue primarily from a static analysis (automatic pool allocation) approach, no work has targeted recycling of capabilities. We have developed a comprehensive extension of our technique assuming a (conservative) garbage collector [3] which makes it possible for our work to handle unbounded heap recycling. Our version-recycling work will be presented in a later publication, separately. Here we only present an interface to the work.

Our novel contributions are given below:

- A table-free method for detecting all memory access errors. Errors covered include uninitialized memory accesses, which are checked in constant time for all types, and not just pointers using a Purify-like technique whose coverage is complete for small allocations and is approximate otherwise. Coverage of all other memory errors is complete within constant time.
- Fat pointers in our method are of scalar sizes, amenable to aggressive optimization, atomic use, and meaningful casts.
- Backward compatibility support is provided extensively by our work, including scalar fat pointers and automatic support for encoding and decoding of pointers.
- Benchmarks of our techniques show that our time overhead for memory-intensive applications averages less than 55%, which is much lower than the nearest prior work.

[1] This comes from the combined counting capacity of heap_offset_bits and version_bits (later), which make up a word.

[2] This in effect is seeking to obtain 32-bit compiled binaries for 64-bit ported library sources where pointers are 64-bit.

2. RELATED WORK

Dhurjati et al. [8] are similar to us in temporal access error checking, although they only cover dangling pointer checks for heap-allocated objects. Our version numbers correspond to virtual page numbers in Dhurjati et al. [8], except that virtual page numbers are shared and looked up via the hardware memory management unit (MMU). While only one version number is generated per allocated object in our scheme, a large object can span a sequence of virtual pages in [8], all of which populate the MMU and affect its performance. Our version numbers are typed by object size and are table-free in terms of lookup. This implies that the object lookup cost is guaranteed to be constant for us, while for [8] it varies according to table size even if OS/hardware supported. For example consider the scenario when the table outgrows the number of pages held in hardware table. TLB misses cost are described as a concern in [8]. There is also concern at the fact that an allocation/deallocation engenders a system call apiece which is expensive.

Our system treats memory violations – temporal and spatial – in an integrated manner. Our versions are substantially more efficient in the virtualization they offer compared to [8] wherein each object allocation, however small, blocks out a full virtual page size and large objects block out multiple virtual pages. By contrast, the virtualization overhead for our mechanism comprises a small constant addition to the object size. Virtual space overuse (simultaneously live objects) has no concomitant performance degradation for us, while in work of [8], it can cause paging-mechanism-related thrashing which would affect not only the application process, but also other processes in the machine.

Xu et al. [22] present a table-based framework to handle temporal and spatial memory access errors. The framework extends the approach of [1] but does not obtain constant-time operations as in our work. Overhead for an allocation operation is linear in the number of pointers to be stored in an allocated block – space for the metadata associated with these pointers is computed and allocated with the block, and initialized as invalid pointers. Also, an allocation request can trigger an expansion of the expandable array store comprising the heap capabilities, which in turn has a linear cost in terms of the total expansion made as the additional slots have to be initialized as the free list of capability slots.

As regards safety checking once pointers have been allocated, not all pointer accesses can be checked in [22] given that pointer metadata in [22] is stored separately from pointers themselves using a source-to-source transformation scheme. Checking safety of a pointer usage requires mirroring access to the pointer by a parallel access to its separately stored metadata, which is not always possible in the approach of [22], which uses statically-exposed access paths (for embedded pointers, page 120, left column, bottom of last paragraph, "worth mentioning …" [22][3]).

[3] Consider the example: struct array {T * zero; long one; T* two; long three;}; struct array * a; T** b = &(a->zero); b += 2; … While [20] mentions on-going work to handle one situation with embedded pointers, it is not clear if that un-reported work would be capable of connecting pointers embedded in b to their metadata since the pointer arithmetic here makes a stride of 2, while the metadata for the struct array makes only a stride of 1 given that no metadata is created for the field one.

The metadata overhead for our fat pointers comprises one extra word at most while the (separately stored) metadata per pointer in [22] comprises two words for capability alone (corresponds to our versions). Additionally, the size of the memory block (referent) pointed to is stored as pointer metadata. Also, an attempt to separate metadata from pointers (i.e not have fat pointers) results in additional overhead of a link field in the pointer metadata. While some of this metadata per pointer gets reduced by sharing it and storing it in the pointed to object, the scheme is unable to reach the shared metadata by pointer arithmetic and ends up having to store an additional pointer to it (* blkhdr, [22] Section 4.1). In [22] the size argument of malloc is used to determine whether an allocation is for an object or array; C programs may use malloc otherwise, which would not work with [22].

Jones et al. [12] present a table-based technique for checking spatial memory violations in C/C++ programs. Standard pointers are used unlike fat pointers of prior spatial access error checkers obtaining significant backwards compatibility as a result. Ruwase et al. [18] extend [12] with out-of-bounds object that allow inbound-pointer-generating arithmetic on an out-of-bounds pointer. Our scalar, fat-pointer based technique has this ability independently of [18, 12].

Dhurjati et al. [7] develop Jones et al. [12] and its extension Ruwase et al. [18] by using automatic pool allocation to partition the large table of objects. The technique statically analyzes application sources. We differ from [7] and its predecessors by not relying on any table lookup. We don't impose any object padding for out-of-bound pointers either. General pointer arithmetic (inbound/out-of-bound) over referent objects is supported by our work.

Loginov et al. [13] present a run-time type checking scheme that tracks extensive type information in a "mirror" of application memory to detect type-mismatched errors. The scheme concedes expensiveness performance-wise (due to mirror costs, not constant time ops – e.g. type information generated is proportional to object size including aggregate objects) and does not comprehensively detect dangling pointer errors (fails past reallocations of compatible objects analogous to Purify).

Purify, by Hastings et al., [10], maintains a map of memory at run-time in checking for memory safety. It offers limited temporal access error protection (not safe for reallocations of deleted data) and fails for spatial access errors once a pointer jumps past a referent into another valid one. Valgrind, [15,19], a dynamic binary instrumentation framework tests for undefined value errors and offers Purify-like protection up to bit-level precision. In contrast to these works, our work captures all dangling pointer errors and spatial errors (e.g. dereference of a reallocated freed object or dereference past a referent into another valid but separate referent). While Valgrind typically slows application performance by well over an order of magnitude, our work adds only limited constant costs to program operations. Also, Valgrind computes some false positives and false negatives within its framework compared to which our approach has no false positives. Our false negatives are limited to uninitialized data checks, wherein our coverage of large objects is approximate.

CCured (Necula et al., [14, 6]) provides a type inference system for C pointers for statically and dynamically checked memory safety. The approach however ignores explicit deallocation, relying instead on Boehm Weiser conservative garbage collection [3] for space reclamation. It also disallows pointer arithmetic on structure fields [14]. The approach creates safe and unsafe pointer types all of

which have some runtime checks. Objects carry size and type tag information. No asymptotic complexities are provided.

Cyclone [11] is a significant enough type-safe variant from ANSI C to require significant porting effort of C programs. In Cyclone, dangling pointers are prevented through region analysis and growable regions and garbage collection. Free() is a no-op, and gc carries out space reclamation. Oiwa's Fail-Safe C [16] uses gc for memory reuse ignoring user-specified memory reclamation.

Berger et al. [2] present a randomized memory manager approach to handling memory safety errors by increasing redundancy (replicating computation; and multiplying heap size, which is similar to Purify's larger heap requirements in support of heap aging). Chilimbi et al. [5] use sample-based adaptive profiling to dynamically build and monitor a heap model, identifying long-unused, stale objects as potential leaks. Our approach can easily replicate this using our list of allocated objects. Further, using the gc extension, this can further guarantee whether an object is a memory leak or not (no pointers left, yet object is live). Qin et al. [17] experiments with using hardware error correcting codes (ECC) in detecting memory violations/leaks in a manner analogous to the page protection mechanism.

3. PROTECTED HEAP MANAGEMENT

Exception protected memory resides in a dedicated heap for the purpose called the protected heap. The stack and global space resides outside the protected heap. Only the protected heap has to have contiguous space reserved for it, which is arranged at the beginning of a program run.

Suppose N is the number of bits used to represent pointers to the address space (i.e. the standard word size, e.g. 64 bits, in a 64-bit architecture). For a protected heap size of 2^M bytes, M is the number of bits needed for addressing bytes in the heap. Then N – M bits remain unused for addressing purposes. These bits can be used for defining version numbers of objects as follows.

A *version* n is the n^{th} time the same object or storage space (as defined by ANSI C99 standard [4]) has been allocated to hold a value.

Storage space is allocated just before the value is constructed and deallocated just after the value is destroyed. Since pointers to an object may survive after the object has been deallocated, the determination that a pointer points to the current object or an earlier version is made using the version bits. The scheme allows 2^{N-M} distinct version numbers, following which version bits must be re-cycled after proving safe recyclability. For a typical 64-bit word machine containing 64-bit pointers, suppose a protected heap of size 4 gigabytes (i.e. 2^{32} bytes) is desired. Then versions totalling $2^{64-32} = 2^{32} = 4G$ in number are supported (after which version recycling needs to be carried out).

We describe our basic technique using C pseudo-code in Figures 1-4. Pseudo-code algorithms are presented, since we argue constant-time complexity of our scheme in Section 6 later. This section ignores alignment considerations for simplicity. Incorporating alignment is discussed separately in Section 4.1. In the figures, $H = 2^{heap\_offset\_bits}$ is the protected heap size. The allocated layout for an object of type T is $L_{sizeof(T)}$, where L_k is defined as given in Figure 1. Note that the layout only involves the size of the type T and not the type itself. Thus the various object lists (Figure 1) manage objects solely by size, and allow storage sharing partitioned by size, not type. In this paper it is assumed that no bitfield is of size 0 (the size 0 cases are straightforward special cases).

```
// presumed:  heap_offset_bits
// presumed void * protected_heap_first
typedef unsigned long word;
#define H (1 << heap_offset_bits)
#define  version_bits  (sizeof(word) * CHAR_BIT – heap_offset_bits)
#define no_of_versions  (1 << version_bits)
word unused_heap_offset = 0;
#define marker_value …  // random
typedef struct
      {word v : version_bits;
       word offset : heap_offset_bits;
      } P;
typedef struct
      {P meta1;   // stores version, next
       P meta2;    // stores marker, previous
       char o[k];
      } L_k;
word allocated_list_k =0;              // initialized empty
word free_list_k = 0;                  // initialized empty
word unusable_free_list_k = 0;     // initialized empty
word last_version_k = no_of_versions – 2;
```

Figure 1. Basic declarations

In this section, we describe our technique for the statically-known size layouts (see Section 4.3 for dynamic sizes). We also use simple, 1-word pointers to access the objects using the encoding for pointers, P, given in Figure 1. For this encoding, a void * pointer is cast to a word P prior to being destructured thus. These simple one-word pointers are incapable of modeling intra-object pointers (to members), which we discuss in Section 4.2 later.

In Figure 1, word is the machine wordsize (e.g. 32 bits, or 64 bits). CHAR_BIT is the number of bits in a byte, ordinarily 8. Our encoded pointers track addresses by the offset (P.offset) from the first location in the protected heap (protected_heap_first). The pointer into the protected heap for unused space is another offset called unused_heap_offset. A marker_value is a random bitmask used for backward compatibility purposes. An encoded pointer P is one word long comprising a version v bitfield and an offset bitfield. The metadata in an object L_k comprises two words, both laid out like P. The first word, meta1 stores the object's version bitfield and an offset to the next object in the linked list that the object belongs to (allocated objects list, free list etc.). The second word, meta2 holds marker_value in its version bits for the purpose of backwards compatibility (discussed later) and an offset to the previous object in the linked list so the object management queues can be doubly-linked lists. All offset fields that point to objects, in pointers, heads of lists below, or in objects themselves always point past the metadata in the pointed object, i.e. to the member o[] in L_k above.

This means a non-empty list has a non-zero head field, allowing 0 to be reserved to indicate an empty list.

For each size k, there are three global lists for managing objects:

- A doubly-linked list of allocated objects which allows any object to be deallocated in constant time. Among other purposes, this list enables encoding of un-encoded pointers returned by un-protected code to provide support for backward compatibility.
- A free list (free_list_k) of previously freed objects that can be used at the next allocation
- An unusable list (unusable_free_list_k) of previously freed objects that can no longer be reused because they have run out of fresh, usable version numbers and require version recycling.

Both the free list and unusable list store objects with the version number advanced to a previously unused version. Thus upon allocation (after recycling – for unusable_free_list_k), this version number can be used directly. Because of this structure, if a dangling pointer test is carried out when a freed object is sitting on one of these two lists, the test will work correctly since the dangling pointer will be encoded with a previously used version while the freed object will have an unused one. The unusable free list is unusable, not because it cannot be allocated from, but because an object allocated by it cannot be freed later (without a preceding recycling).

Without recycling, versions would be allocated in increasing, round-robin order from 0 till no_of_versions – 1, where the last version is reserved for residence on the unusable list. While a full treatment of recycling is not in the scope of this paper, it suffices to say that the upper limit of version allocations also wraps around and moves within the range [0 … no_of_versions -1]. The limit separates freed version numbers from versions that may still be in use. This limit is tracked by last_version_k that moves round-robin in the range of version numbers. It is initialized to no_of_versions – 2 since at the start, no recycling is involved, and no_of_versions – 1 is reserved for the unusable list whose objects and object pointers are known to not be in use.

```
// translate version-carrying pointer
void * decode_pointer (void * ptr)
 { return protected_heap_first +  ((P *) &ptr)->offset; }
bool verify (void * ptr) {
  return ((P *) &ptr) ->v ==
       (((P *)  decode_pointer(ptr)) – 2)  ->v;}
```

Figure 2. Read/write related operations

An encoded pointer is translated to standard C pointer in Figure 2 by obtaining the offset field within the pointer and adding that to protected_heap_first. Pointer decoding precedes each dereference of the object. Prior to decoding the pointer thus, memory safety check requires that the version stored in the object be consistent with the version stored in the pointer. This can be carried out by verify, wherein the right hand side of the equality test carries out the former and the left hand side carries out the latter.

The allocation procedure is statically customized to size k (prefix k in allocate_protected_k). First an attempt to allocate from the free list is made. If that fails, then an attempt to allocate from the unused heap is made. In this attempt, the version assigned is taken to be two past the (rotating) last_version_k limit. As mentioned earlier, one past the last_version_k is number reserved for the unusable free list. If allocation does not succeed from either free list or unused heap, then an allocation failure is indicated by returning NULL. NULL is a constant, encoded pointer to a constant, never-deleted, zero-sized object (i.e has no o[] field) allocated in the protected heap at the beginning of program execution. While checking against NULL can be treated as a special case check to be added explicitly to the verify operation in Figure 2 above, this check is gracefully merged into usual spatial error checking in Section 4.2. Returning NULL indicates allocation failure.

Allocation creates and populates an encoded pointer (ptr) with the pertinent offset and version v fields. Once an object to allocate is obtained, meta points to the start of the metadata affiliated with the object. Finally the object metadata is modified to reflect the doubly-linked structure of allocated_list_k. The *previous* offset field of any existing head object is set to the newly allocated object; the newly allocated object's *previous* is set to 0 reflecting its position at the head; the head points to the newly allocated object and the newly allocated object's *next* points to the previous head object.

```
void * allocate_protected_k () {
 P * ptr, meta;
 if (free_list_k != 0)   // allocate from free list
    { ptr->offset = free_list_k;
      meta = ((P *) (protected_heap_first + free_list_k)) – 2;
      ptr->v = meta->v;
      free_list_k = meta-> offset; }
  // allocated from unused heap
 else if (H – unused_heap_offset >=  k + 2 * sizeof(P) ) {
    ptr->offset = unused_heap_offset + 2 * sizeof(P);
    meta = (P *) (unused_heap_offset + protected_heap_first);
    meta->v =  (last_version_k + 2) % no_of_versions;
    (meta + 1)->v = marker_value;
    unused_heap_offset = unused_heap_offset + k + 2 * sizeof(P) ;
    ptr->v = meta->v;}
 else return NULL;  // indicates failure to allocate
 meta->offset = allocated_list_k;
 if (allocated_list_k != 0)            // set previous
    (((P *) (protected_heap_first + allocated_list_k)) -1)->offset
           =  ptr->offset;
 allocated_list_k = ptr->offset;    // set head of list
 (meta + 1) -> offset = 0;  // set previous
 return *((void **) ptr);  /* return encoded pointer  */ }
```

Figure 3. Allocation

Deallocation is also customized to size k. In Figure 4, it is presumed that verify is executed beforehand to verify version and non-NULL legality.

A successful deallocation increments (via increment_version_k) the version of the object that can be used both while sitting on a free list or by the next allocation. In incrementing if it is found that the last_version_k limit is crossed, then the object is placed on the unusable free list, otherwise it is placed on the standard free list. The crossing of last_version_k is decided by computing the gap between the current version and the limit. Suppose last_version_k ≥ meta->v. Then gap in Figure 4 should be last_version_k - meta->v, which is indeed the case as the modulo arithmetic drops the no_of_versions addition. Suppose last_version_k < meta->v. Then gap in Figure 4 should be last_version_k + (no_of_versions - meta->v), which again is the result offered by the modulo arithmetic. Thus Figure 4 computes the correct gap in all cases.

```
bool increment_version (P * meta, word last) {
  word gap = (last + (no_of_versions – meta->v))  % no_of_versions;
  meta->v = (meta->v + 1 )  % no_of_versions;
  return (gap != 0);
}
void deallocate_protected_k (void * ptr) {
  P * meta = ((P *)  decode_pointer(ptr))  – 2 ;
  word next = meta->offset;
  word previous = (meta + 1) -> offset;
  // modify object version and add to appropriate free list
  if (increment_version(meta, last_version_k))
     { meta->offset = free_list_k;
       free_list_k = ((void *) (meta + 2)) – protected_heap_first;}
  else {meta->offset = unusable_free_list_k;
     unusable_free_list_k = ((void *) (meta + 2)) – protected_heap_first;}
  // remove from allocated list
   if (previous == 0)  allocated_list_k = next;    // reset next
   else (((P *) (protected_heap_first + previous)) – 2)->offset = next;
   if (next != 0)
      (((P *) (protected_heap_first + next)) – 1)->offset = previous; }
```

Figure 4. Deallocation

4. GENERAL POINTER AND LANGUAGE CONSIDERATIONS

4.1 Alignment Issues

Type alignment can be built in simply by allocating objects along the most general alignment, doubleword boundaries. Figure 5 shows the allocation layout for an object of type T.

```
struct L {word marker1 : version_bits;
          word size : heap_offset_bits;
          word v : version_bits;
          word next: heap_offset_bits;
          word marker2: version_bits;
          word previous: heap_offset_bits;
          word init;                          // initialization flags
          doubleword[ ⌈sizeof(T)/sizeof(doubleword)⌉ ] o;}
```

Figure 5. Aligned object layouts

The space cost of rounding sizeof(T) up to a multiple of doubleword can be reduced directly to a multiple of word. However, the allocation interface would then become different from the standard one for malloc(), which only takes object size as the argument and not alignment.

The size field in Figure 5 allows spatial safety checks to be carried out (Section 4.2). Bitfields marker1 and marker2 are used to place fixed bit patterns in the protected heap to aid backward compatibility searches (Section 5).

Another departure the layout in Figure 5 makes over the simpler layout in Figure 1 is that an init field is kept for the purpose of un-initialized access checks in the allocated object. This is a Purify-like approach wherein the object is divided into equal areas, each represented by an initialization flag. A write sets the area's init flag. If a read is carried out in an area before the init bit is set, it indicates an un-initialized field access. This approach captures un-initialized reads of all types, and not just pointer types as is obtained in [1, 22]. Furthermore, since the number of flags is a constant, the initialization checks (e.g. resetting flags upon object allocation) all transpire in constant time unlike the linear-in-object-size cost of [1, 22]. For small objects, the flags cover initialization errors comprehensively, for large objects, the coverage is approximate.

One of the useful features of this arrangement is that all meta data for object o wastes no padding bits or bytes and minimally occupies four words before member o. Furthermore there is no padding after member o if its alignment is doubleword. The stored object size in an object's metadata omits the padding incurred by the field o in rounding to a doubleword. This is for the purpose of accurate spatial checks.

4.2 General Arithmetic-Supporting Pointers

In C/C++, pointers are scalar types so they ought to be represented within one or two machine words (consistent with standard scalar sizes). Figure 6 presents our general encoded pointers in two words.

```
struct P^G {
  word v : version_bits; word offset : heap_offset_bits; // word 1
  word  intra_object_pointer;                            // word 2
}
```

Figure 6. General pointer layout

In the above layout of general pointers (P^G), the first word encodes version-carrying pointer data as discussed in the algorithms presented earlier (Figures 1-4). The second word stores a regular un-encoded pointer that can point to any inner member of the object. This pointer is not stored as an offset and occupies a whole word so that following C's semantics, general pointer arithmetic can shift this pointer around the whole machine address space (within and beyond protected heap) without bothering whether the pointer points to a valid object or not. Thus C's general pointer arithmetic is fully supported (it is carried out directly on the un-encoded pointer). Validity checking occurs only when a pointer is dereferenced, to check whether the object pointed to is inbound or not.

A reduced-heap implementation of our pointer is given in Figure 7. This implementation is pertinent when the heap requirements of a program are small. In the context of migration of 32-bit programs to 64-bit platforms, even the largest useful heap sizes can well be the largest supported by 32-bit systems. The doubled size of a 64-bit pointer means that meta-data beyond the bits needed for addressing the largest 32-bit heaps can be stored within one 64-bit pointer. The meta-data stored beyond heap_offset_bits required to address the heap is optimized by converting the one-word occupying intra_object_pointer from Figure 6 to an intra_object_offset in Figure 7. This conversion is based on the insight that an intra-object pointer is likely to mostly remain inbound (this drives the work in Jones et al. [12], where mostly, further reach of the pointer is explicitly disallowed).

The number of bits required to represent intra_object_offset is computed by the following static analysis. The maximum size of an object allocated by the program is estimated (this is typically known from the associated type in case of non-array objects). The size is bounded by the protected heap size, which can further be bounded more tightly by the user in which case a dynamic bounds check each time an object is allocated is carried out. The maximum deviation of a pointer out-of-bounds is estimated. For this, it is known that the maximum deviation by pointer arithmetic can only occur prior to a dereference using the pointer. The dereference dynamically checks for the pointer being inbound. Each pointer if properly initialized, is initialized as inbound or a NULL pointer wherein the intra_object_offset is zero[4]. Proper initialization is verified statically in our work for now. The maximum that a pointer can deviate beyond this inbound or zero offset into invalidity is bound by the largest chain of pointer arithmetic operations that can be executed in the program *before* a dereference of the pointer. A static proof that each pointer arithmetic operation must be succeeded within a finite path by a dereference of the pointer is sufficient to bound the maximum deviation. The deviation is the maximum sum of the pointer offsets carried out along any such path in the program. This is carried out intra-procedurally in our work as this seems to be quite sufficient so far.

Once the maximum bound on any pointer's outbound excursion is computed, intra_object_offset_bits is computed as $1 + \log_2$ (maximum excursion bound + maximum allocated object size). If the maximum excursion bound is not a known constant, the reduced heap implementation is not used[5]. The extra bit is required for the sign bit to cover negative offsets.

```
struct P^R {
  signed word  intra_object_offset : intra_object_offset_bits;
  word v : version_bits;
  word offset : heap_offset_bits;
}
```

Figure 7. Reduced-heap pointer layout

The object layout for a reduced-heap implementation changes from Figure 5 to include padding equivalent to intra_object_offset field in each of the first three meta-data words.

Spatial test for a reduced-heap pointer comprises casting its intra_object_offset to an unsigned word and checking whether it is less than the unsigned object size. This is a fast *one-comparison test* (instead of conjunction of two tests for upper and lower bounds), in which negative offsets are always larger than any object size due to the contribution of the sign bit (note that size is represented in heap_offset_bits which are always fewer than a word due to version_bits). Spatial test for a general-heap pointer uses the same test as above, after generating an intra_object_offset equivalent from the offset and the intra_object_pointer fields.

As described in Section 3, the NULL pointer is encoded to point to an object of size 0, which means that its spatial test will always fail. This is a special object containing only meta-data fields. NULL pointer dereferences are caught as spatial errors during dereferences, which eliminates special-case treatment. For a free operation, it is checked that the intra-object-offset is 0 besides the regular spatial and temporal checks.

Pointer arithmetic operations are modified to increment or decrement the intra_object_offset or intra_object_pointer fields in an encoded pointer. Note that this maintains pointer arithmetic operations as constant-time operations.

4.3 Statically Unknown Allocations

Given that C's malloc takes a dynamic size argument, the search of the corresponding object lists (or allocation/deallocation functions as described here) is a dynamic cost. While for the large majority of cases, the dynamic size would be tied to a (statically-known) type's allocation (hence sizeof() is known statically), a user is free to allocate space completely dynamically (e.g. one of the benchmarks here, MST, allocates an array of size provided by user input dynamically). For the former case of the statically known types, the search can be eliminated statically as described in Figures 1-4. For the unusual, fully dynamic case, the search cost can be bounded to a constant in our scheme as follows: Dedicate a P^{th} portion (P is a constant) of the protected heap of size H as a search array to contain access data for all dynamic sizes handled by the heap. Memory other than the protected heap can be used for this purpose. The p^{th} slot in the array can contain access data for sizes [p*P … (p +1)*P -

[4] Note that a pointer can be created using a cast from integer explicitly or implicitly in which case the pointer's outbound excursion cannot be assumed to be zero unless the novel support provided by our work here (see backwards compatibility, section 5) in mapping the integer to an inbound pointer or NULL is relied upon.

[5] A user-defined bound on intra_object_offset_bits can still be used, with dynamic checking carried out at each pointer arithmetic operation to optionally implement a reduced heap strategy.

1] within itself that can be searched in time proportional to constant P. In effect this search array provides a hash table with constant search size (clash per bucket). This method may be fine-tuned based on static/dynamic profiling/analysis of information of the sizes actually generated by the program.

4.4 Stack and Globals Protection

Any stack scalar variable requiring run-time protection checks for the storage it represents (e.g. an automatic variable whose address is taken) is shifted to the heap. This is straightforwardly done by wrapping the variable's type in a struct. An automatic variable initialized by the struct allocation is then generated so that every time it is instantiated in a new stack frame, the struct is heap allocated. References to the original scalar are replaced by references to the automatic variable's struct member. Each time the stack frame is destroyed, the structs allocated for its variables are de-allocated so that no later dereferences are allowed. At the time the stack frame is destroyed, the pointers to the allocated structs are checked for liveness as a part of deallocation. If any of the structs has been deallocated before, then an exception is thrown, which catches the user-deallocation of stack variables.

Similarly, a global scalar requiring run-time protection is also moved to heap by replacing it with an un-initialized struct-wrapped counterpart and changing global references in the program to the struct member. The user-defined main() is renamed and called from within a system-generated main() that initializes the global structs with allocated objects. The system-generated main() deallocates the global structs at its end, whereupon user deallocations of globals are caught.

Non-scalar automatic and global variables are handled similarly, without requiring wrapper structs.

5. BACKWARD COMPATIBILITY

As mentioned in the introduction, the scalar sizes of our fat pointers can enable them to be compiled at the same size as standard pointers. Backward compatibility then reduces to the problem of providing encoding and decoding support for pointers when interacting with unprotected code through libraries, pointer casts to integers etc. Of these the decoding problem is simple; the interface code walks over the data to be passed to unprotected code and calls decode_pointer (Figure 2) and replaces encoded pointers with decoded pointers in place (in data). The NULL pointer is decoded to the standard C NULL pointer as a special case. The problem of encoding non-NULL pointers passed back from un-protected code is more involved and is as described below.

First the allocation functions linked to un-protected/library code are made variants of the protected heap allocation functions as follows. The allocator returns protected heap objects on request, with the change that a decoded pointer to the object is returned, and not an encoded pointer. Prior to returning the object, the decoded pointer and its encoded version are stored in a global table for use later by interface functions.

Once the unprotected/library code finishes executing and the interface to the code is reached, all data returned by the unprotected code is walked in order to replace decoded pointers by encoded ones. The global table populated by allocations above is used as an association list in this replacement process as is the set of decodings that were carried out when the unprotected code was entered.

The association list of encoded/decoded pointers cannot suffice in general. For the decoded pointers whose encoding is still not found, the following method is used. From the location pointed by the decoded pointer in protected heap, a preceding pair of marker1 marker2 patterns is located in the heap. A sanity check that these are indeed intended marker values is carried out by traversing the previous and next fields relative to the markers to locate their objects and corresponding marker values. Consistency check with these objects increases confidence in the pattern discovery. In searching for the preceding markers of a decoded pointer, only preceding memory up to the size of the largest-allocated object has to be searched. The search starts from the nearest preceding marker pair such that the associated size field keeps the decoded pointer within bounds of the associated object. For each such candidate object, the previous object in the doubly-linked list of objects is looked up. Each shift to a previous object is checked for consistency with a traversal back using the next link. If a consistent traversal back to an allocated_list_k header is obtained, only then it is assured that the starting marker values represent a valid, live object. Once the validity of the object containing the decoded pointer is verified, then the encoded pointer is generated straightforwardly. It is assumed that for non-NULL pointers, the unprotected code only returns pointers intended to be inbound and to live objects. If no live containing object is found, then an error is reported.

An integer cast to pointer generates an undecoded pointer initially, which is then converted to an encoded pointer as discussed above. Similarly encoded pointers are cast to integer by first converting them to decoded pointers.

6. PERFORMANCE

As far as the asymptotic performance of our algorithms is concerned, note that none of the routines in Figures 1-4 (and their general pointer discussion, Section 4) have any loops or recursion. Any search cost for object lists/accessors for any object size k is constant as described using P-denominated structures in Section 4.3. Thus the cost for providing memory safety (allocation, deallocation, pointer arithmetic, and verification overhead) in our system comprises only constant time operations.

In this section we characterize the cost constants of our work. For this, we have both reduced-heap and general implementations run on a 64-bit machine (AIX 6.1.0.0, Power5 2.09GHz, 4G RAM) using GCC 4.2.4 for compilation at –O3 level of optimization, with version recycling/garbage collection within our system completely disabled. We have benchmarked our performance on the memory-intensive applications of the Olden Suite which comprises programs that have been commonly benchmarked by the relevant related work. We have benchmarked only publicly available Olden applications (all that we could find, which was from the Cyclone site, containing four Olden applications in all, see http://www.cs.umd.edu/projects/PL/cyclone/benchmarks-1.0.tar.gz). The benchmarks contain several NULL-dereference errors, all of which were caught by our work. For the benchmarks, the general and reduced-heap implementations were chosen such that all encoded pointer bitfields are rounded to multiples of a byte. This enables a specialized kernel to be generated in which bit-field access gets replaced by field access and pointer arithmetic in general.

The general-heap benchmarks use one-byte version_bits and four-byte heap_offset_bits, wasting three-bytes as padding. The reduced-heap implementation uses one-byte version_bits, three-byte

intra_object_offset_bits, and four-byte heap_offset_bits. Due to the lack of 128-bit integer types in GCC (encoding as a 128-bit long double runs into a GCC bug at –O3 level optimization), we split the 128-bit general-heap-encoded pointer into two 64-bit unsigned long quantities (one the intra_object_pointer and the other containing the version_bits and heap_offset_bits). The two longs are carried everywhere the original pointer is, as scalars using a straightforward source-to-source transformation. When storing the pointer in memory, or communicating with the external world, the two longs are placed adjacent to each other just as they would be in a 128-bit layout (Figure 6).

The static analysis (for intra-object-offset field size, Section 4.2) establishing these benchmarks to be capable of reduced-heap implementation also establishes proper initialization, which means that the run-time initialization check mechanism is eliminated from these benchmarks. Furthermore, no stack or global variables require heap-shifting (as none of them involve arrays, or have their address taken). These optimizations are commensurate with the optimizations carried out in [1, 8, 22]. While [1, 22] do incur an extra dynamic overhead of resetting any pointers in allocated memory blocks, this cost is minor (resetting allocated blocks to 0 adds less than 0.2% to original application times). Hence the cost comparison is generous, since [1] has additional run-time optimizations enabled eliminating expensive temporal checks dynamically and [8] uses a combination of static and run-time methods in automatic pool allocation to reduce run-time costs. Our results are shown in Table 1 and contrasted with prior work. The column unprotected run time gives the average time taken by an application for one run in a batch of twenty runs. The times are measured using getrusage() system call and comprise the user + system times. The cost of setting up the protected heap using an sbrk() call is included in each application's time.

Among our benchmarks, MST performed the worst, in part because it accesses the kernel via the P-denominated structures of Section 4.3. This is because MST dynamically allocates arrays of a size that is provided as user input. Hence allocations for these arrays become dynamically-sized and the kernel access acquires a layer of dynamic deconstruction described in Section 4.3. In the reduced heap case, we benchmarked the application using fixed array sizes also. This reduced the overhead down to 83%, an improvement of 11.3% that brings the average overhead of reduced heap implementation to below 49% (for programs which do not have dynamically-sized mallocs).

Note that on average, our work performs better than the nearest prior work [22] by a factor of 2.33 for general heap and 2.42 for reduced heap. We report comparisons with [1, 22] since they share our goals of complete memory safety for C without changing the memory model (free() not obviated by garbage collection). We have also considered [8] since its temporal checking via virtual pages is close to our own core concept of versions. Other approaches that we haven't contrasted with individually here have different goals than us (changed memory model – CCured [6, 14], Cyclone [11], and Fail-Safe C [16]; or address a subset of safety issues (mostly spatial) – Jones et al. [12], Ruwase and Lam [18], Dhurjati et al. [7] and Loginov et al. [13]). To the best of our knowledge, our work advances the state of the art in complete memory safety for C-like languages by well over a factor of 2 (see comparison with [22] above).

7. CONCLUSION AND FUTURE WORK

We have presented a novel scheme for comprehensive safety in the context of memory models like that of the C language. We do not change the memory model (e.g. free() remains meaningful, and not subsumed by garbage collection). Instead, we provide safety for all issues along with the following novel results: (a) our work is table free, saving time/space costs of lookup; (b) operations like deference checking overhead, allocation, deallocation, and pointer arithmetic overhead have only constant-time costs; (c) encoded pointers are fat, but within scalar sizes (one to two words), which makes them amenable to aggressive optimization, backward compatibility and atomic use; (d) backward compatibility now has support for encoding and decoding of arbitrary pointers; (e) benchmarks show good performance even on memory-intensive Olden applications (< 55% average overhead, over twice faster than nearest prior work), which suggests that our work is likely to be even faster in the usual, not-so-memory-intensive applications.

We plan to complete our on-going implementation of conservative garbage collection for version recycling and to report our experience on a larger set of benchmarks.

8. ACKNOWLEDGEMENTS

Harshit Shah was supported by an Indo-Italian Fellowship under the ITPAR Programme. He has carried out this work at IBM India Research Laboratory, New Delhi.

Table 1. Benchmarks of Olden Suite Applications

Benchmark	*Unprotected run time (seconds)*	*Reduced Heap (overhead, %)*	*General Heap (overhead, %)*	*[1] (overhead, %)*	*[8] (dangling ptr checks only, overhead, %)*	*[20] (overhead, %)*
TREEADD	1.00	15	38	-	268	223
MST	0.35	94	98	400	853	76
BISORT	3.17	49	57	-	222	76
TSP	2.90	49	24	-	312	128
Average	1.86	52	54	400	414	126

9. REFERENCES

[1] Austin, T. M., Breach, S. E., and Sohi, G. S. 1994. Efficient detection of all pointer and array access errors. In Proc. ACM SIGPLAN 1994 Conf. Programming Language Design and Implementation (Orlando, Florida, United States, June 20 - 24, 1994). PLDI '94. ACM, New York, NY, 290-301. DOI=http://doi.acm.org/10.1145/178243.178446.

[2] Berger, E. D. and Zorn, B. G. 2006. DieHard: probabilistic memory safety for unsafe languages. In Proc. ACM SIGPLAN 2006 Conf. Prog. Language Design and Implementation,, SIGPLAN Not. 41, 6 (Jun. 2006), 158-168. DOI=http://doi.acm.org/10.1145/1133981.1134000.

[3] Boehm, H. 1993. Space efficient conservative garbage collection. In Proc. ACM SIGPLAN 1993 Conf. Prog. Language Design and Implementation (Albuquerque, New Mexico, United States, June 21 - 25, 1993). R. Cartwright, Ed. PLDI '93. ACM, New York, NY, 197-206. DOI=http://doi.acm.org/10.1145/155090.155109.

[4] ISO/IEC 9899:1999 C standard, 1999. ISO/IEC 14882:1998 C++ standard, 1998. Also, ISO/IEC 9899: 1999 C Technical Corrigendum, 2001, www.iso.org.

[5] Chilimbi, T. M. and Hauswirth, M. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In ASPLOS 2004, SIGPLAN Not. 39, 11 (Nov. 2004), 156-164. DOI=http://doi.acm.org/10.1145/1037187.1024412.

[6] Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W. 2003. CCured in the real world. In Proc. ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (San Diego, California, USA, June 09 - 11, 2003). PLDI '03. ACM, New York, NY, 232-244. DOI=http://doi.acm.org/10.1145/781131.781157.

[7] Dhurjati, D. and Adve, V. 2006. Backwards-compatible array bounds checking for C with very low overhead. In Proc. 28th Int. Conf. Software Engineering (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 162-171. DOI=http://doi.acm.org/10.1145/1134285.1134309.

[8] Dhurjati, D. and Adve, V. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In Proc. Int. Conf. Dependable Systems and Networks (June, '06). DSN '06. IEEE Computer Society, Washington, DC, 269-280.

[9] Dhurjati, D., Kowshik, S., and Adve, V. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In Proc. ACM SIGPLAN 2006 Conf. Prog. Language Design and Implementation, SIGPLAN Not. 41, 6 (Jun. 2006), 144-157. DOI=http://doi.acm.org/10.1145/1133255.1133999.

[10] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In Proc. Usenix Winter 1992 Technical Conference (San Francisco, CA, USA, Jan. 1992). Usenix Association, 125-136.

[11] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. 2002. Cyclone: A Safe Dialect of C. In Proceedings of the General Track: 2002 USENIX Annual Technical Conference (June 10 - 15, 2002). C. S. Ellis, Ed. USENIX Association, Berkeley, CA, 275-288.

[12] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In Automated and Algorithmic Debugging, Linkoping, Sweden, pages 13--26, 1997.

[13] Loginov, A., Yong, S. H., Horwitz, S., and Reps, T. W. 2001. Debugging via Run-Time Type Checking. In Proc. 4th International Conf. Fundamental Approaches To Software Engineering (April 02 - 06, 2001). H. Hußmann, Ed. LNCS vol. 2029. Springer-Verlag, London, 217-232.

[14] Necula, G. C., McPeak, S., and Weimer, W. 2002. CCured: type-safe retrofitting of legacy code. In Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Portland, Oregon, January 16 - 18, 2002). POPL '02. ACM, New York, NY, 128-139. DOI= http://doi.acm.org/10.1145/503272.503286.

[15] Nethercote, N. and Seward, J. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (June 2007). PLDI '07. ACM, New York, NY, 89-100. DOI= http://doi.acm.org/10.1145/1273442.1250746.

[16] Oiwa, Y. Implementation of a Fail-Safe ANSI C Compiler. PhD Thesis, Department of Computer Science, University of Tokyo, December 2004.

[17] Qin, F., Lu, S., and Zhou, Y. 2005. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In Proc. HPCA (February 12 - 16, 2005). IEEE Computer Society, Washington, DC, 291-302.

[18] Ruwase, O. and Lam, M. 2004. A practical dynamic buffer overflow detector. In Proc. Network and Distributed System Security (NDSS) Symposium. February 2004, 159-169.

[19] Seward, J. and Nethercote, N. 2005. Using Valgrind to detect undefined value errors with bit-precision. In Proc. USENIX Annual Technical Conference (Anaheim, CA, April 2005). USENIX '05. USENIX Association, Berkeley, CA.

[20] Varma, P. "Generalizing Recognition of an Individual Dialect in Program Analysis and Transformation", In Proc. ACM Symp. Applied Computing (SAC 2007) (Seoul, Korea, March 11-15, '07) ACM Press, New York. 1432-1439. DOI=http://doi.acm.org/10.1145/1244002.1244310.

[21] Varma, P. Anand, A., Pazel, D. P., Tibbitts, B. R. "NextGen EXtreme Porting: Structured by Automation", In Proc. ACM Symp. Applied Computing (SAC 2005) (Santa Fe, NM, USA, March '05) ACM Press, New York. 1511-1517. DOI=http://doi.acm.org/10.1145/1066677.1067018.

[22] Xu, W., DuVarney, D. C., and Sekar, R. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In Proc. 12th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (Newport Beach, CA, USA, October 31 - November 06, 2004). SIGSOFT '04/FSE-12. ACM, New York, NY, 117-126. DOI= http://doi.acm.org/10.1145/1029894.1029913.