Flexible and Efficient Memory Object Metadata

Zhengyang Liu
Beijing University of Posts and Telecommunications, China
zhengyang-liu@hotmail.com

John Criswell
University of Rochester, USA
criswell@cs.rochester.edu

Abstract

Compiler-based tools can protect software from attack and find bugs within programs. To support programs written in type-unsafe languages such as C, such tools need to add code into a program that must, at run-time, take a pointer into a memory object and locate metadata for that memory object. Current methods of locating metadata are either flexible (supporting metadata of varying sizes) at the expense of speed and scalability or are fast (e.g., by using shadow tables) at the cost of flexibility (metadata is small and must always be the same size).

This paper presents a new method of attaching metadata to memory objects, named *Padding Area MetaData* (*PAMD*), that is both flexible and efficient. Metadata can be any size, and different memory objects can have different sized metadata. While flexible, the algorithm for finding the metadata given a pointer into the memory object takes constant time. Our method extends Baggy Bounds with Accurate Checking (BBAC) which attaches constant-sized metadata to memory objects for performing precise dynamic bounds checks. Our design supports variable-sized metadata, and our implementation supports larger programs.

We evaluated the performance and scalability of PAMD using dynamic bounds checking as an exemplar of our method. Our results show that our method adds at most 33% overhead to an identical dynamic bounds checking tool that trades precision for performance by using a simple shadow table. Our results also show that our method, while having the same flexibility as splay trees, performs significantly faster and scales better as a program allocates more memory.

CCS Concepts • **Software and its engineering** → **Allocation / deallocation strategies**; *Software testing and debugging*; • **Security and privacy** → Software and application security

Keywords memory metadata, memory safety, security hardening, shadow table, dynamic analysis

1. Introduction

Compiler-based tools can protect software from attack [11, 12] and find bugs within programs [8, 29, 30]. Some of these tools need to add code into a program that must, at run-time, locate metadata for the memory object to which a pointer points i.e., find metadata for the pointer's *referent memory object* [18]. Applications of such techniques include dynamic array bounds checking [11, 12, 18], memory error checkers [5, 30], dynamic type error detectors [20], dynamic information flow tracking [21], and dynamic data race detection [8].

There are two methods of attaching such metadata to memory objects. The first approach is to use one or more splay trees [31] to record metadata about each memory object [11, 12, 18, 21]. Splay trees are flexible; they can record different-sized metadata for memory objects. However, splay trees grow larger as the program allocates more memory objects; our experiments in Section 5.4 show that the time needed to search through the splay tree increases as the number of memory objects increases. Splay tree methods are impractical for programs manipulating large amounts of data, and while there are optimizations that create multiple smaller splay trees that can each be searched more quickly [12], such optimizations require sophisticated inter-procedural points-to analysis.

A second approach is to create a shadow table; memory is divided into small contiguous constant-sized groups called *slots* [6], and each slot has an entry in the shadow table. A pointer to a memory location can be converted into an index in the shadow table in constant time [6]. Since shadow tables make locating metadata fast, numerous tools [6, 8, 26, 30] use them. However, because shadow tables have an entry for each slot, the metadata is the same size for each memory object, and tools employing shadow tables often sacrifice flexibility, functionality, or accuracy to keep shadow table entries small. For example, SharC [8] limits the number of concurrent threads it can check for data race detection to keep its shadow table small. Memory safety tools such as Address Sanitizer [30] maintain one bit of metadata and determine whether a pointer points into an allocated memory object; they cannot determine to which memory object a pointer should point or how far outside the referent memory object an out-of-bounds pointer has wandered.

To build more sophisticated software debugging and protection tools, developers need a method of attaching metadata to memory objects that is *flexible* (permitting variable-sized metadata), *efficient*, and *scalable*. A previous workshop paper proposed BBAC [14]: a method of performing precise dynamic bounds checks by storing metadata about the precise bounds of memory objects inside the internal fragmentation created by an existing and less accurate bounds checking system named Baggy Bounds Checking (BBC) [6]. BBC aligns and pads memory objects to ensure that memory references stay within the bounds of the expanded object. BBAC places bounds information within the padding area added by BBC. While the previous workshop paper demonstrated that BBAC adds little overhead over BBC [14], its results are limited: the original BBAC prototype only worked on a few programs, and the evaluation did not study the causes of overhead and the scalability of the approach. More importantly, the original BBAC work only addressed dynamic array bounds checking.

We generalize the BBAC approach in a new method named *Padding Area MetaData* (*PAMD*): *any* metadata that a tool needs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ISMM'17, June 18, 2017, Barcelona, Spain
ACM. 978-1-4503-5044-0/17/06...$15.00
http://dx.doi.org/10.1145/3092255.3092268

to attach to a memory object can be stored within the padding area and located at run-time using a few simple bitwise and arithmetic operations on pointers which point into the memory object. Like BBAC and BBC, our method locates metadata in constant-time. Finally, PAMD can attach metadata of different sizes to different memory objects, providing flexibility.

We have created a more robust implementation of BBAC in the open-source SAFECode compiler [3] built using LLVM [22], extended it to create the PAMD prototype, and evaluated its performance on the Olden [27] and SPECint benchmarks [15], the GNU Zip and Bzip2 compression programs, the Sqlite3 database server, and the CPython interpreter. Our evaluation shows that the additional cost of accessing our metadata adds, at most, 33% overhead over the existing costs of using BBC within the SAFECode compiler. Additionally, our evaluation determines the source of the overheads incurred by PAMD; it also shows that PAMD scales well as programs allocate more memory objects to process more data.

To summarize, our contributions are:

- A method of attaching metadata to memory objects that allows run-time checks to locate the metadata of a pointer's referent memory object in constant time. Furthermore, this method does not require the metadata to be small or constant-sized.
- An efficient and robust implementation of the PAMD prototype with new optimizations that were not employed by BBAC [14]. Our implementation is memory allocator neutral; it works with any memory allocator that allows programs to specify the alignment of memory objects.
- An evaluation of PAMD that shows that the additional time spent reading the metadata located in the padding area is small compared to current approaches that only use a specialized direct lookup table.
- An evaluation showing which components of our design induce the most overhead.
- An evaluation showing that PAMD provides better performance and scalability for tracking memory object metadata than splay trees without sacrificing functionality. To the best of our knowledge, this is the first scalability study for dynamic array bounds checking.

The rest of this paper is organized as follows. Section 2 provides background on Baggy Bounds Checking. Section 3 presents the design of PAMD, and Section 4 describes our PAMD implementation. Section 5 describes our performance evaluation of PAMD. Section 6 discusses related work, Section 7 discusses future work, and Section 8 concludes.

2. Background: Baggy Bounds Checking

Baggy Bounds Checking (BBC) [6] is a referent object-based solution for dynamic array bounds checking. In referent object checking, when a run-time check needs to determine whether a pointer variable holds an address within an allocated memory object, the check performs a lookup to find the base and bounds of the memory object containing that address [18]. Jones and Kelly introduced the first referent object checking approach [18] for dynamically checking pointer arithmetic; given a pointer into a memory object, their system ensures that any pointer derived from a computation on the first pointer e.g., an array indexing operation, points to the same object. Jones and Kelly used a single splay tree [31] to store and locate memory object metadata [18], causing overheads of $5\times$ to $6\times$.

To reduce performance overhead, BBC [6] changes the memory allocation of memory objects to allow it to use a shadow table. BBC employs a buddy allocator which pads every memory object

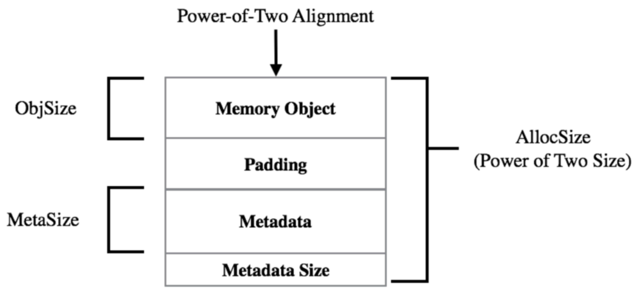


Figure 1. Variable-Sized Metadata in Padding Area

to a power-of-two length, denoted $AllocSize$, and aligns its start address to be a multiple of $AllocSize$. This allocation strategy allows BBC to represent the size and alignment of each memory object using a single byte representing the binary logarithm of the allocation size of the memory object.

To record the bounds information about memory objects, BBC allocates a contiguous array that it uses as a bounds table (denoted as $table$ or as the "baggy bounds table") [6]. BBC partitions the virtual address space into equal-sized *slots* each of size $slot\_size$ bytes. All memory allocation sizes are a multiple of $slot\_size$. The bounds table contains an entry for each slot; each entry contains the binary logarithm of the allocation size of the memory object occupying the slot. To look up bounds information for a pointer's referent memory object, BBC determines the slot into which the pointer points (a simple bit-shift operation) and looks up the entry for the slot in the table. With the binary logarithm of the allocation size and alignment (denoted e), BBC can quickly compute the base and allocation size of the memory object using Equations 1 and 2:

$$base = p \,\&\, \sim (2^e - 1) \tag{1}$$

$$AllocSize = 2^e \tag{2}$$

BBC only performs dynamic array bounds checking [6] and has limited support (or no support) for detecting other memory safety errors such as dangling pointers, integer overflow errors, type-casting errors, and uninitialized pointer errors. However, compared to other tools that perform dynamic array bounds checking, BBC performs well: BBC [6] is about twice as fast as the best splay tree approach [12] (which employs multiple splay trees). However, BBC trades precision for speed: it cannot determine whether a pointer lands in the padding area or within the bounds of the original memory object allocated by the program [6]. While acceptable for security protection, this limits BBC's utility in testing and debugging tools.

3. PAMD Design

Our design of PAMD extends BBC [6] to attach metadata to memory objects and allows run-time checks to find the metadata associated with a pointer's referent memory object in constant time. Unlike the original BBAC system presented in the workshop paper [14], our PAMD design supports both variable-sized and constant-sized metadata. As Figure 1 illustrates, our new method places the metadata attached to a memory object (in the case of the original BBAC, the original size of the memory object [14]) and the length of the metadata at the end of the padding area added by BBC. For tools that use metadata of a constant size, the metadata size field can be omitted. The BBC lookup can quickly locate both the *beginning* and the *end* of the memory object. Once a run-time

Algorithm 1: Register Memory Object

```
Input: $Base$: First address of memory object
Input: $MetaSize$: Size of Metadata in bytes
Input: $ObjSize$: Size of original memory object
Function Register (Base, ObjSize, MetaSize)
    begin
        $Index \leftarrow p \gg log_2(SlotSize)$;
        $AllocSize \leftarrow$
          $CalculateAllocSize(ObjSize, MetaSize)$;
        $e = log_2(AllocSize)$;
        $range \leftarrow 1 \ll (e - SlotSize)$;
        $memset(BaggyBounds + Index, e, range)$;
        $Metadata \leftarrow Base + range - MetaSize$;
        $Initialize(Metadata)$;
    end
```

Algorithm 2: Unregister Memory Object

```
Input: $p$: Pointer to deallocated memory object
Function Deregister (p)
    begin
        $Index \leftarrow p \gg log_2(SlotSize)$;
        $AllocSize \leftarrow BaggyBounds[Index]$;
        if $AllocSize\ != 0$ then
            $range \leftarrow 1 \ll (AllocSize - SlotSize)$;
            $memset(BaggyBounds + Index, 0, range)$;
        end
    end
```

Algorithm 3: Lookup Metadata

```
Input: $p$: A pointer
Output: $Metadata$: Pointer to Metadata
Function Lookup (p)
    begin
        $Index \leftarrow p \gg log_2(SlotSize)$;
        $e \leftarrow BaggyBounds[Index]$;
        if $e == 0$ then
            return NULL
        end
        $AllocSize \leftarrow 1 \ll e$;
        $Base \leftarrow p \wedge (AllocSize - 1)$;
        $MetaSizeAddr \leftarrow$
          $Base + AllocSize - pointersize$;
        $MetaSize \leftarrow *(MetaSizeAddr)$;
        $Metadata \leftarrow Base + AllocSize - MetaSize$;
        return Metadata
    end
```

check computes the last address of the memory object, it can locate the length of the metadata stored at the end and compute the location of the metadata. In essence, PAMD exploits the additional padding that BBC adds to memory objects to store the metadata and the metadata's size.

3.1 Memory Allocation

Like BBC [6] and BBAC [14], PAMD divides the virtual address space into *slots* each of $SlotSize$ bytes in size and uses a contiguous array of one-byte bounds information with one entry per slot.

Figure 1 shows a memory object and its metadata. Before a memory object is allocated, the size must be increased to include the size of the memory object allocated by the program (denoted $ObjSize$), the size of the metadata (denoted $MetaSize$) and the size of a pointer (if variable-length metadata is used). We call this size $AdjustedSize$. If $AdjustedSize$ is less than the size of a single slot, it is increased to the size of a slot. The size must then be increased to the next power-of-two size that is equal to or greater than $AdjustedSize$; we denote this as $AllocSize$.

After allocation, PAMD must register the bounds of the new memory object in the baggy bounds table and initialize the memory object's metadata. Algorithm 1 shows the procedure. It first computes the index in the table for the first slot occupied by the new memory object. It then computes the number of slots occupied by the memory object (denoted $range$) and sets all the table entries to the exponent of the size/alignment of the newly allocated memory object. Once the entries in the baggy bounds table are initialized, a function to initialize the metadata of the memory object (and the metadata size field for variable-sized metadata) is called.

The baggy bounds table must be updated when a memory object is freed. To unregister a memory object, Algorithm 2 uses the baggy bounds table to locate the allocation size of the memory object ($AllocSize$) and then locates all of the entries in the baggy bounds table for the memory object and sets them to zero. If the pointer does not point into a memory object, then $AllocSize$ will be zero, and Algorithm 2 does nothing.

3.2 Metadata Lookup

With PAMD's allocation strategy, metadata lookup takes constant time. Algorithm 3 takes a pointer p and locates the metadata of the pointer's referent memory object. It first right-shifts p by the constant $log_2(SlotSize)$ to get the index in the bounds table, denoted by $Index$. It then retrieves the entry in the bounds table which records the binary logarithm of the allocation size (denoted by e) and computes the allocation size of the memory object (denoted by $AllocSize$). Once $AllocSize$ is computed, the run-time check can easily find the first and last address of the memory object. The size of the metadata is stored within the last word of the memory object, and the start of the metadata precedes the last machine word stored in the padded memory object. If all metadata is of the same size i.e., $MetaSize$ is constant, then the beginning of the metadata is at $Base + AllocSize - MetaSize$.

If a pointer has no referent memory object i.e., it points into memory in which there is no memory object, then the bounds table will contain a zero. Algorithm 3 returns a $NULL$ metadata pointer in such a case.

4. Implementation

We updated the open-source SAFECode compiler [3, 13] to LLVM 3.7 and modified it to use PAMD for its run-time checks; we also improved the BBC-like approach from the BBAC work [14]. The SAFECode compiler transforms programs at the LLVM IR level. By default, it does not use the multiple splay tree or automatic pool allocation optimizations [12, 13] and is therefore essentially an implementation of the Jones and Kelly approach [18] with the Ruwase and Lam [28] extension for handling out of bound pointers that are never dereferenced. We used the SAFECode compiler as it inserts run-time checks on memory accesses and pointer arithmetic operations[1] and will therefore stress the performance of our PAMD approach. SAFECode's metadata is constant-sized, and our PAMD prototype takes advantage of this by storing the memory object's actual size at the end of the padding area as BBAC [14] does. We

[1] The original BBC [6] only checks array indexing; SAFECode checks additional operations to provide more comprehensive memory safety.

disabled this optimization by adding code to the run-time checks to perform a volatile load and an add; this mimics the overhead of reading the metadata's size from the end of the memory object. Our implementation works on the 64-bit x86 architecture on the Linux 3.16.0 and FreeBSD 9.3 operating systems.

4.1 Baggy Bounds Table

Linux and FreeBSD support 2^{47} user space addresses per process on 64-bit x86, starting at virtual address 0 [16, 25]. We chose a slot size of 16 bytes (identical to BBC [6]) and divided the user space addresses into 2^{43} slots. Each memory slot has a corresponding entry in the baggy bounds table, resulting in a table that occupies 8 TB of virtual address space. When a program begins execution, it uses `mmap()` to allocate space for the baggy bounds table. Unlike the original BBC system [6], our implementation does not hard-code the location of the baggy bounds table but instead allows the operating system kernel to place the table where it prefers; the implication is that our instrumentation must load the address of the baggy bounds table from a global variable on each run-time check whereas the original BBC does not induce this overhead. As `mmap()` does not map physical memory to the mapped virtual range until the virtual address is accessed [9, 25], our bounds table occupies physical memory proportional to the number of entries in the bounds table that are modified during program execution.

4.2 Transforming Memory Allocation

We added four new transformation passes to the LLVM compiler that modify memory allocations to pad and align memory objects as PAMD requires. Our compiler pads and aligns heap, stack, and global memory objects as well as call by value (byval) arguments. Figure 2 summarizes how the transforms appear in the LLVM IR.

Heap Allocations For heap allocations, we created a customized wrapper library around the system's heap allocator; our prototype provides wrappers for `malloc()`, `calloc()`, `realloc()`, and `free()`. We implemented a wrapper for `malloc()` named `__bb_malloc()`. When `__bb_malloc()` is called, it first computes the allocation size ($AllocSize$) of the heap object as Section 3 describes and then calls `posix_memalign()` to allocate the object with the proper alignment. It then initializes the metadata section at the end of the padding area. Finally, it initializes the baggy bounds table entries for the slots occupied by the new memory object.

When freeing a heap object, the baggy bounds table entries for slots comprising that object must be zeroed. We therefore created a wrapper for the `free()` function named `__bb_free()` which uses Algorithm 2 to zero the baggy bounds entries and then calls the system `free()` function to release the memory.

The `calloc()` function works similarly to `malloc()`; the difference is that `calloc()` takes two arguments and uses the product of these two arguments as the heap object size. To pad, align, and register memory objects allocated by `calloc()`, we created a wrapper function for `calloc()`. It multiplies the arguments and then calls the customized `malloc()` with the product.

Given a pointer and a new size, the `realloc()` function reallocates a new area of memory. To ensure that the old allocation is correctly freed and the new allocation is correctly prepared, we created a wrapper function for `realloc()`. The wrapper function first calls `__bb_free()` to free the old memory, uses `__bb_malloc()` to allocate new memory, and then copies data from the old memory object to the new memory object. While less efficient than a custom `realloc()` implementation, this approach works regardless of how the system allocator is implemented.

An LLVM pass rewrites all calls to the heap allocation functions to call the customized allocator/deallocator functions.

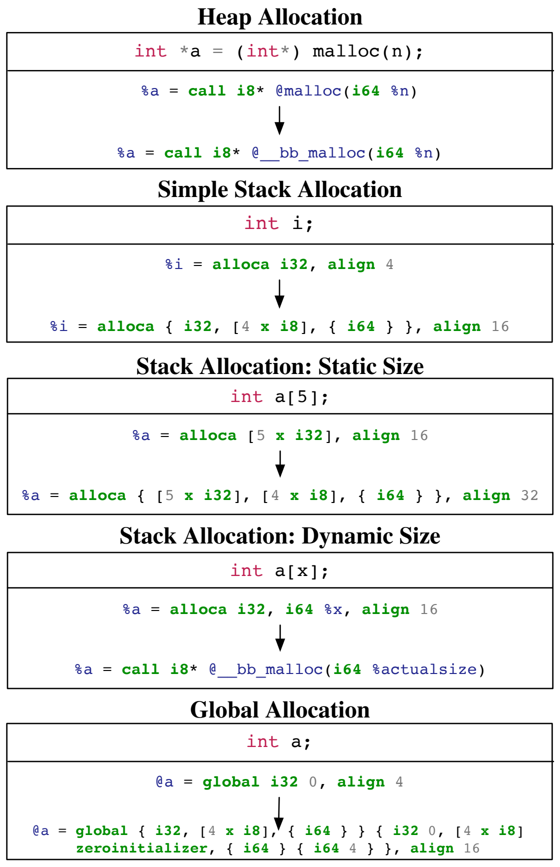

Heap Allocation
```
int *a = (int*) malloc(n);
```
```
%a = call i8* @malloc(i64 %n)
%a = call i8* @__bb_malloc(i64 %n)
```
Simple Stack Allocation
```
int i;
```
```
%i = alloca i32, align 4
%i = alloca { i32, [4 x i8], { i64 } }, align 16
```
Stack Allocation: Static Size
```
int a[5];
```
```
%a = alloca [5 x i32], align 16
%a = alloca { [5 x i32], [4 x i8], { i64 } }, align 32
```
Stack Allocation: Dynamic Size
```
int a[x];
```
```
%a = alloca i32, i64 %x, align 16
%a = call i8* @__bb_malloc(i64 %actualsize)
```
Global Allocation
```
int a;
```
```
@a = global i32 0, align 4
@a = global { i32, [4 x i8], { i64 } } { i32 0, [4 x i8]
     zeroinitializer, { i64 } { i64 4 } }, align 16
```

Figure 2. Transformations

Stack Allocations LLVM provides the `alloca` instruction to allocate memory on the stack [22]. Stack allocations can either have a static size known at compile-time or can compute an allocation size dynamically during execution. We wrote one pass that finds statically sized `alloca` instructions and replaces them with new `alloca` instructions that allocate stack memory with the adjusted size and proper alignment. Another pass transforms `alloca` instructions with dynamically computed allocation sizes into heap allocations. This is needed because the `alloca` instruction takes a constant value for its alignment operand; alignment operands for stack-allocated memory cannot be computed at run-time [23].

Global Variables Our prototype processes global variables similarly to stack objects whose size can be computed at compile time. An LLVM pass transforms the global variables and adds padding to global variables with initializers.

Call by Value (byval) Arguments The LLVM IR marks call by value function arguments with the `byval` attribute; the code generator transparently generates code that allocates memory for byval arguments and copies the data from the caller's copy to the callee's copy [23]. To correctly handle these arguments, the compiler pass to transform byval arguments creates a clone of the function that has the byval argument properly aligned and padded as needed by PAMD; the pass will also add code to the beginning of the clone to configure the baggy bounds table entries for the byval argument and to initialize the byval argument's metadata.

The compiler pass updates direct calls to the original function to call the new clone. The original function is left within the program but is modified to call the new clone, thereby allowing functions with byval arguments that are visible to external code to work properly with PAMD.

4.3 Runtime Checks

The open-source SAFECode compiler, upon which our prototype is based, is designed to provide comprehensive protection against memory safety errors. As a result, it adds run-time checks to more operations than other memory safety tools. Below, we describe the run-time checks that SAFECode inserts into programs.

- **Getelementptr Checks**: The SAFECode compiler adds checks after every pointer arithmetic operation (performed by the `getelementptr` instruction [22]) to ensure that pointer arithmetic does not create an out-of-bounds pointer. Checks are performed on pointer arithmetic for array indexing as well as pointer arithmetic for computing the offsets of fields within a structure (if the memory object for the structure was incorrectly allocated with too small a size, structure pointer arithmetic can generate an out-of-bounds pointer).
- **Load/Store Checks**: The SAFECode compiler adds run-time checks before every load, store, and atomic read/modify/write instruction. These checks check for dangling pointers and catch the use of invalid pointers that occur through means other than errant pointer arithmetic e.g., bad casts, incorrect use of C unions, etc. These checks also ensure that the first and last byte of a memory read or write stays within the bounds of a single memory object.
- **Out of Bound (OOB) Pointer Rewriting**: If a `getelementptr` check determines that a computed pointer is out of bounds, it replaces its value with an out-of-bounds (OOB) pointer value; this value will cause a trap if used in a memory access instruction [28]. This method of handling OOB pointers is less efficient than the method used in the original BBC system [6] but places fewer constraints on how the virtual address space of the process is organized.

Our prototype does not instrument calls to external C library functions such as `strcpy()` and `memcpy()` as this requires enhancing our PAMD run-time library with specialized run-time checks for these functions. We will add this enhancement in future work.

4.4 Safety Transforms

Since we extended the open-source SAFECode compiler to prototype our approach, our prototype utilizes these additional memory safety transformations:

- **Stack Allocation Initialization**: Initialize pointers within stack allocated objects so that use of uninitialized pointers causes a segmentation fault.
- **Invalid Free Checks**: Verify that pointers to deallocation functions point to the beginning of allocated memory objects.
- **Control Flow Integrity Checks**: Add checks before indirect function calls to ensure that they call one of several functions as identified by call graph analysis. Unlike solutions that *only* enforce control flow integrity [4], checks on function returns are not required as the memory safety checks prevent corruption of return addresses on the stack [11, 13].

4.5 Optimizations

We employ several optimizations within the SAFECode compiler to make our implementation more efficient. These include:

1. **Fast Check Conversion**: When a run-time check is performed within the same function in which a memory object is allocated, there is no need to look up the memory object's metadata; the check on the pointer can use the bounds information available within the function. A similar optimization is performed for checks on global variables.
2. **Run-time Check Elimination**: Run-time checks on pointer values that are used in comparisons but never used for memory accesses can be eliminated.
3. **Run-time Check Inliner**: By default, the SAFECode compiler inserts calls to run-time checks that are implemented within an external library. The run-time check inliner pass inlines the code for the run-time check within the compiler, alleviating the overhead of calling the run-time checks without using link-time optimization.
4. **Memory Object Registration Inliner**: Like run-time checks, the SAFECode compiler inserts calls to an external library function to modify the baggy bounds table when memory objects are allocated or deallocated. We have modified the compiler to inline this code as well.

5. Experimental Results

In this section, we address some of the key questions about PAMD:

- What is the cost of PAMD adding metadata to memory objects?
- What are the sources of overhead in PAMD?
- How well does PAMD scale as programs process more data?
- How large can metadata reasonably be when using PAMD?

We evaluated PAMD's performance on the SPECint 2006 benchmarks in the SPEC CPU® 2006 benchmark suite [15], the Olden benchmarks [27], GNU Zip, Bzip2, SQLite3, and the C Python interpreter.

5.1 Hardware Setup

We ran our SPECint 2006 experiments on a shared multi-user server with dual Intel Xeon® E5-2630 v3 processors at 2.40 GHz with 32 GB of RAM. Each processor has 8 cores, 16 threads, and 20 MB of cache. We ran the other experiments on a dedicated single-user server with an Intel Xeon® E5-2609 processor at 2.40 GHz with 4 cores, 4 threads and 10 MB of cache, 16 GB of RAM, and a RAID 0 array with dual 15,000 RPM 6 Gb/s 600 GB SAS disks managed by an IBM ServeRAID M5110e RAID controller. Both of the machines run the Linux 4.8.15 SMP kernel. All measurements are the average of ten runs; we report the sum of user and system time. We compiled all programs with `-O3` optimization.

5.2 Cost of Metadata

We wanted to determine the overhead of the additional memory accesses needed to read the metadata used for run-time checks. Therefore, we compared the performance difference of PAMD to our own implementation of BBC [6] in the SAFECode compiler. Our BBC implementation checks whether a pointer is within the allocation size instead of checking the pointer against the precise object bounds stored within the metadata; this essentially replaces SAFECode's run-time checks on loads, stores, and pointer arithmetic with "baggy" versions that pass if the checked pointer falls either within the original memory object or the padding area added by BBC. Our BBC and PAMD implementations use the same optimizations and check the same operations; the only difference is the use of the metadata. Section 4.3 provides more details on the SAFECode run-time checks. For these experiments, we used the SPECint benchmarks, the Olden benchmarks, GNU Zip, Bzip2, SQLite3, and CPython.

SPECint 2006 We tested BBC and PAMD on the SPECint 2006 benchmarks. For SPECint 2006, we disabled Out-of-Bounds

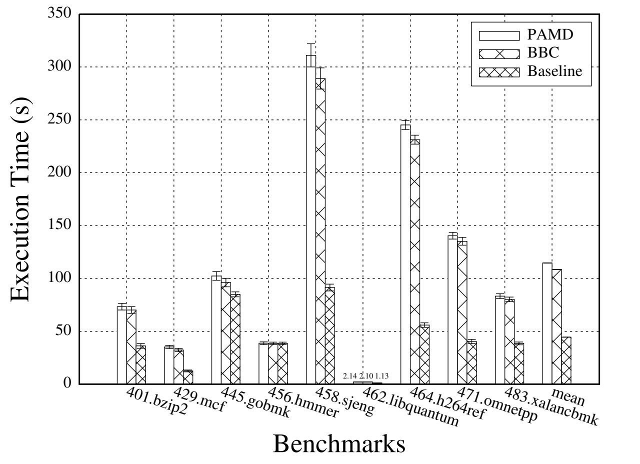


Figure 3. SPECint 2006 Benchmarks

(OOB) pointer rewriting. Since we used the LLVM test suite to run the SPECint benchmarks, all of the benchmarks use the `test` input except for 400.perlbench and 462.libquantum which use the `train` input. Figure 3 shows the results for the SPECint 2006 benchmarks; the baseline is the execution time of programs compiled by the SAFECode compiler with the same level of optimization (`-O3`) but with memory safety transformations disabled so that it compiles code as the original LLVM compiler does. We do not show results for 400.perlbench and 473.astar because BBC and PAMD detect memory safety errors in these programs; we are still investigating the memory safety error reports to see if they are caused by bugs in our prototype or genuine bugs in the benchmarks. A bug in BBC and PAMD prevent us from compiling 403.gcc.

As Figure 3 shows, nearly all of the SPECint 2006 benchmarks which work with our memory safety compiler introduce negligible overhead when switching from BBC to PAMD; the differences between the averages are within a standard deviation. The exception is 464.h264ref which shows a 25% increase in execution time, relative to the baseline, when switching from BBC to PAMD.

Olden We expanded the input size of each Olden benchmark to the values shown in Table 1 as the original input sizes are too small for modern machines. Since the `Power` benchmark does not require any input, we execute it 30 times and use the total time as the result of a single run. Figure 4 shows the results of running both BBC and PAMD on the Olden benchmarks with all of our optimizations enabled. Our results show that the additional time for reading the metadata from memory adds very little additional overhead. While the maximum difference of average execution time between PAMD and BBC is 12%, Figure 4 shows that the difference between PAMD and BBC is within a standard deviation of the averages.

GNU Zip and Bzip2 We then evaluated PAMD by compressing files with gzip-1.8 and bzip2-1.0.4. We used the Linux pseudorandom number generator device `/dev/random` to create files with random contents of sizes ranging from 5 MB to 1 GB.

For gzip-1.8, we measured the execution time of compressing files of sizes 50 MB to 1 GB using PAMD and BBC. For bzip2-1.0.4, we did the same but used files of size 5 MB to 100 MB. Figures 5 and 6 report the average of ten runs for each size. Our results show that BBC and PAMD perform similarly, though for larger files, the cost of PAMD reading the extra metadata increases. The percent increase in latency, relative to the baseline, for switching from BBC to PAMD is 16% for gzip and 19% for bzip2.

Benchmarks	Input size
BH	60000 20
Bisort	4000000
Em3D	1024 850 125
Health	10 50 2
Mst	7000
Perimeter	11
Power	not available, loop 30 times
TreeAdd	24 1
Tsp	2048000
Voronoï	1000000 20 32 7

Table 1. Input Size Used in Olden Benchmarks

SQLite3 We also compiled Sqlite3 using the LLVM compiler, BBC, and PAMD. We selected 8 benchmarks from the Speed benchmark [1] contained in the Sqlite3 source tree and expanded the problem size to get execution times of over 10 seconds with BBC and PAMD. Table 2 briefly describes the inputs for Sqlite3 and the increased input sizes. As Figure 7 shows, both BBC and PAMD incur significant overhead over the baseline program compiled with the LLVM compiler. However, it also shows that the overhead of PAMD is within the standard deviation of BBC.

Benchmark ID	Input size
1	100000 INSERTSs
2	250000 INSERTs in a transaction
3	100 SELECTs without an index
4	100 SELECTs on a string comparison
5	200000 SELECTs with an index
6	500 UPDATEs without an index
7	250000 UPDATEs with an index
8	250000 text UPDATEs with an index

Table 2. Inputs for Sqlite3 Experiments

CPython Finally, we compiled CPython-2.7.12 to evaluate PAMD on an interpreter. We chose eight benchmarks from the Python performance benchmarks [2]. We compared the execution time of PAMD with BBC on these benchmarks. The results in Figure 8 show that the difference in performance between PAMD and BBC is within a standard deviation with the exception of `nqueens` (which shows a 33% increase in latency relative to the baseline)

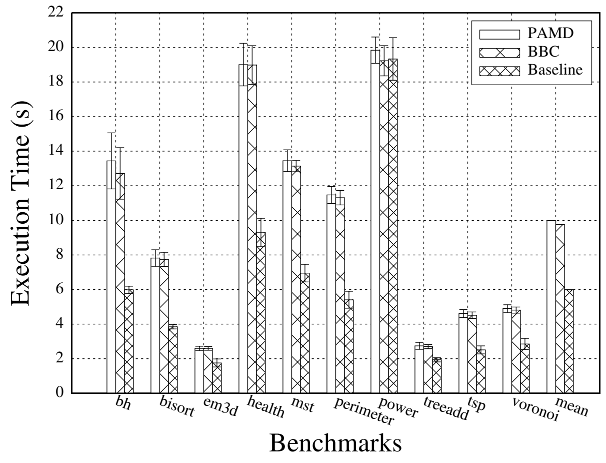


Figure 4. Olden Benchmarks

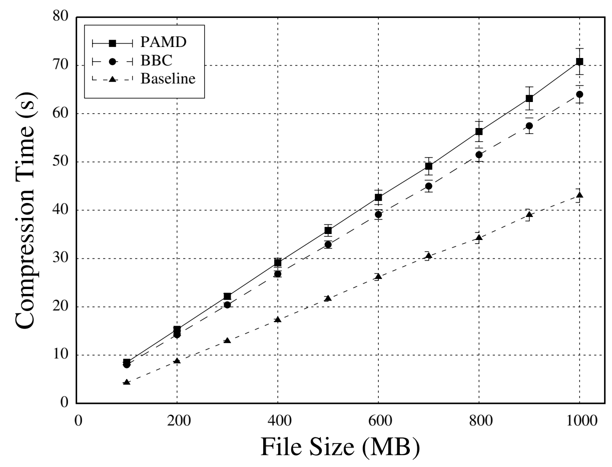


Figure 5. Gzip-1.8 Performance

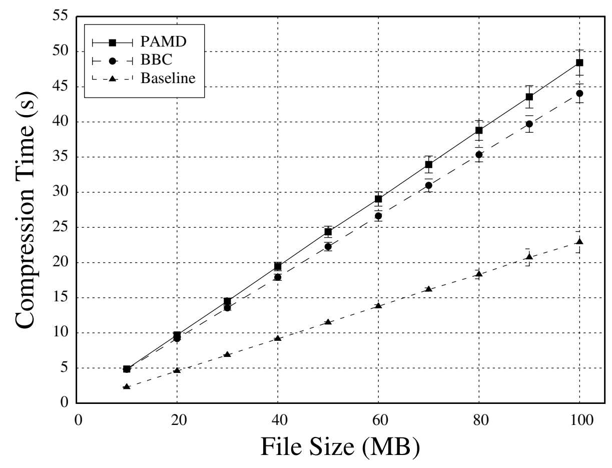


Figure 6. Bzip2-1.0.4 Performance

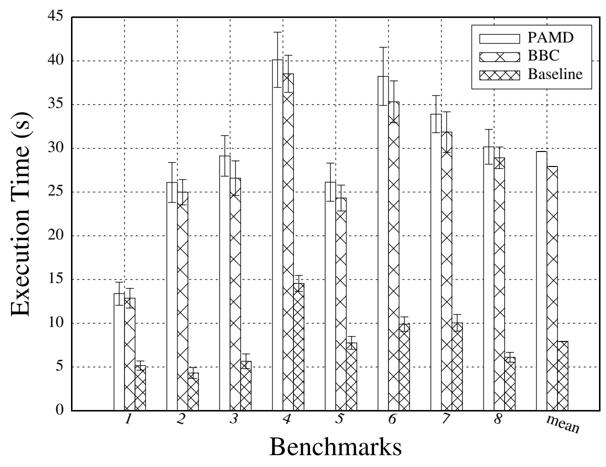


Figure 7. Sqlite3 Performance

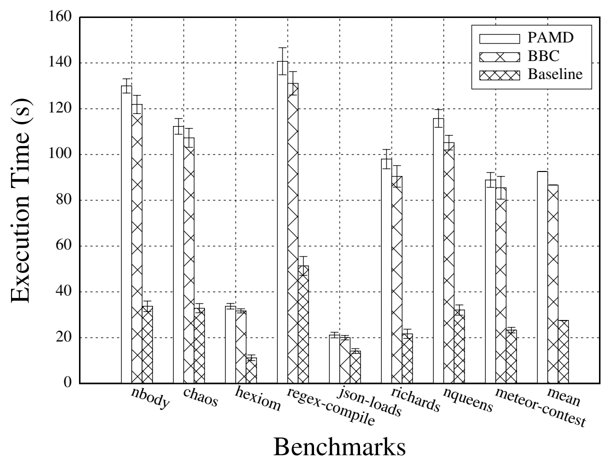


Figure 8. CPython-2.7.12 Performance

and nbody (which shows a 24% increase in latency relative to the baseline).

5.3 Sources of PAMD Overhead

PAMD incurs three primary sources of overhead (abbreviations are in parentheses):

- **Allocation Change** (**AllocChange**): The overhead incurred by changing the alignment and allocation size of memory objects
- **Object Registration** (**ObjReg**): The time spent modifying the baggy bounds table when allocating and freeing memory objects
- **Runtime Check** (**Check**): The time spent reading the memory object's metadata and performing the bounds check

To measure these overheads, we disabled all the compiler transform passes that add code to the program to perform each of the aforementioned operations, ran the Olden benchmarks, and then progressively added each transformation back. These experiments therefore show how much overhead is added for each of the above operations.

The results, shown in Figure 9, indicate that most of the overhead incurred by PAMD is due to initialization of the baggy bounds table and the metadata lookup/run-time checks: object registration adds the most overheads to four of the Olden benchmarks while the run-time checks add the most overhead for five other benchmarks.

Increasing the slot size could reduce object registration overhead. Compiler optimizations could also eliminate some object registrations if they can prove that a run-time check will never return the address of a memory object; this can occur, for example, when a memory object is allocated within a function and only ever used within that function. Aggressive static analysis (such as static array bounds checking and redundant run-time check elimination) may be able to eliminate run-time checks from correctly written code.

5.4 Scalability Experiments

Splay trees and PAMD can both record variable-sized metadata for memory objects. However, each makes different tradeoffs. The time to insert and locate items in a splay tree depends on the number of items in the tree [31]; the amount of memory allocated should not affect splay tree performance, but allocating more memory objects will degrade splay tree performance. PAMD, on the other hand, can locate metadata in constant-time but must initialize more entries in the baggy bounds table as the amount of memory allocated and freed grows. The number of memory objects should not affect PAMD performance, but allocating more memory can degrade PAMD performance.

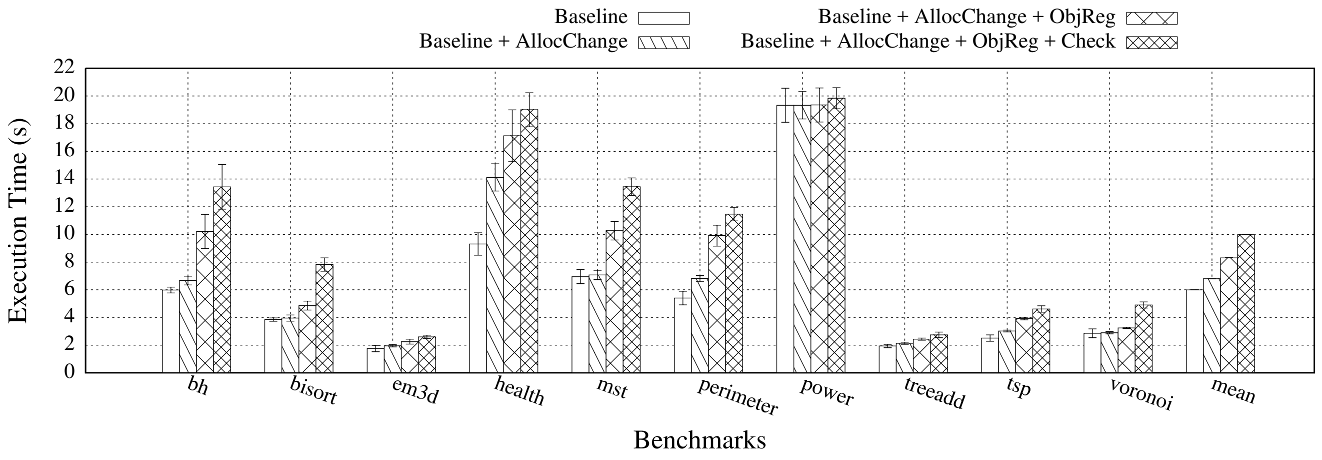


Figure 9. Overhead Analysis for Olden Benchmarks

We compared the performance of PAMD to splay trees on the Olden benchmarks. We then measured the scalability of splay trees and PAMD using several benchmark programs from our previous experiments for which the input size can be varied to make the program do more computation. For each program, we measured the peak memory usage and maximum number of memory objects allocated to understand its memory allocation behavior and then measured the performance of the program when performing memory safety checks with splay trees and PAMD for increasing input sizes. To summarize, our two memory usage metrics are:

- **Peak Allocation Size**: Maximum number of bytes allocated for memory objects (global, stack, and heap) at any point in time during a single execution
- **Peak Object Count**: Maximum number of memory objects allocated at any point in time during a single execution

For these experiments, we disabled run-time check inlining as it is unavailable for splay trees. We also disabled the code to mimic the overhead of reading the metadata size field as the splay tree implementation assumes constant-sized metadata.

Olden We first compared the execution time overhead of PAMD with the single splay tree implementation in the open-source SAFECode compiler using the Olden benchmarks. We disabled the inlining optimization in PAMD as the splay tree implementation lacks this optimization; we also disabled the volatile load to mimic variable-sized metadata. As Figure 10 shows, PAMD performs much better than a single splay tree, incurring 8.3× overhead on average while the splay tree method incurs 51× overhead.

Python nqueens We compiled CPython and ran the `nqueens` benchmark from the Python performance benchmark suite [2]; `nqueens` takes a positive integer N as input and finds all possible solutions of the N-Queens problem using a brute-force algorithm.

To determine the relation between the amount of data processed during the execution and the input size, we first measured the peak allocation size and object count with input N ranging from 3 to 10. Figure 11(a), which uses a logarithmic scale on the y-axis to better show small values, reports the results and shows that `nqueens` allocates more memory and more memory objects as its input size grows. We then measured the execution time of `nqueens` using PAMD and splay trees; the baseline is the unmodified program compiled with the LLVM compiler.

Figure 11(b) reports the average execution time normalized to the baseline under different input sizes. The results show that the

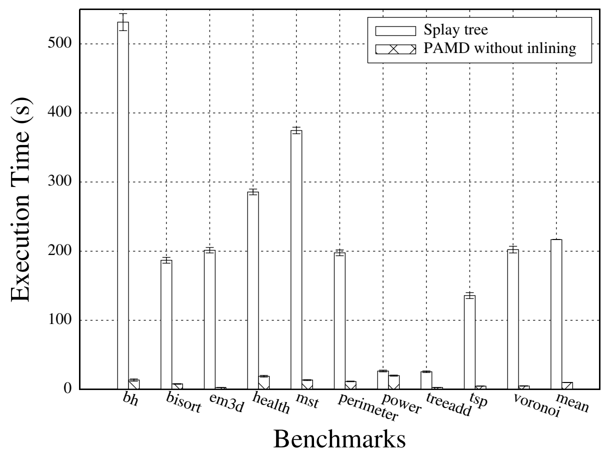


Figure 10. Comparison of Splay Tree to PAMD

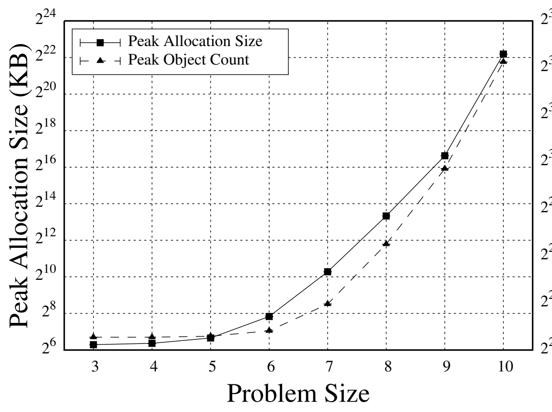
normalized splay tree time increases from $68\times$ to $156\times$ as the program input grows while PAMD scales better with the normalized time ranging from $2.6\times$ to $3.2\times$.

GNU Zip and Bzip2 We also evaluated the scalability of PAMD and splay trees on gzip-1.8 and bzip2-1.0.6 using the same files as in Section 5.2. Figures 12(a) and 13(a) report the peak allocation size and object count for each input size for gzip and bzip2, respectively, while Figures 12(b) and 13(b) show the gzip and bzip2 execution times normalized to the respective baselines when compressing files of increasing size.

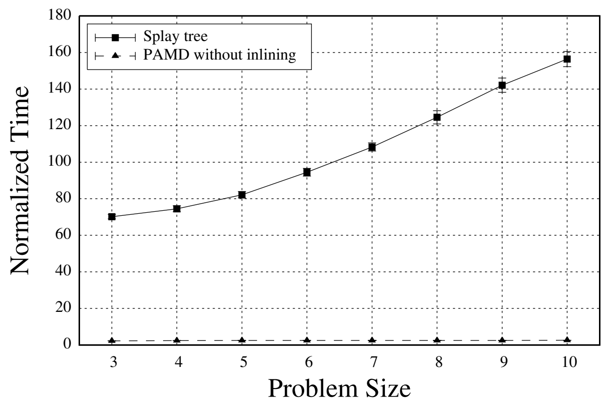
Gzip does not allocate more memory or more memory objects as its input size grows. As a result, the overhead of the splay tree and PAMD implementations stays relatively constant as the input size increases. In contrast, Bzip2 allocates more memory and more memory objects as it compresses larger files. As a result, the splay tree implementation exhibits higher overhead as the input size increases (starting at $60\times$ and growing to $84\times$). However, PAMD overhead remains relatively constant (from $2.1\times$ to $3\times$).

5.5 Metadata Size

Since bounds checking can use constant-sized metadata, our prototype omits the word-sized metadata length field in Figure 1 and uses the same 8-byte field to store the original object length; our

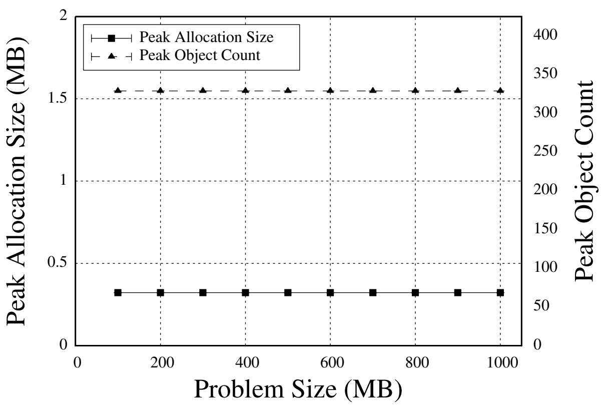


(a) Nqueens Allocation Size and Object Count vs Problem Size

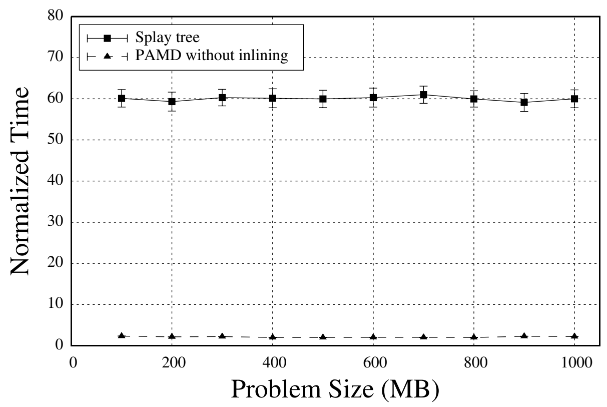


(b) Nqueens Execution Time vs Problem Size

Figure 11. Nqueens Scalability Results



(a) Gzip Allocation Size and Object Count vs Problem Size



(b) Gzip Execution Time vs Problem Size

Figure 12. Gzip Scalability Results

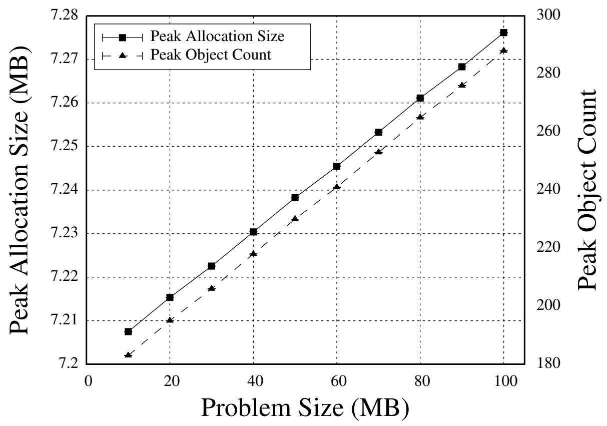
PAMD prototype essentially uses the minimal amount of metadata space that our approach can allocate. Other tools will need to use more of the padding area to store their metadata. How large can the metadata get before memory objects will need to be expanded to the next power-of-two size to accommodate larger metadata?

We measured the average number of unused padding bytes for each allocation size between 2^5 and 2^{12} bytes in size for the SPECint Benchmarks, GNU Zip, and Bzip2 to determine how large metadata could be before needing to expand memory object size any larger. As Figure 14 shows, on average, memory objects have at least 8 additional bytes of space for metadata. As many tools use one byte or less of metadata [6, 8, 30], we believe that 8 bytes of metadata will help improve the precision of existing tools and enable the development of more sophisticated debugging and security hardening tools.

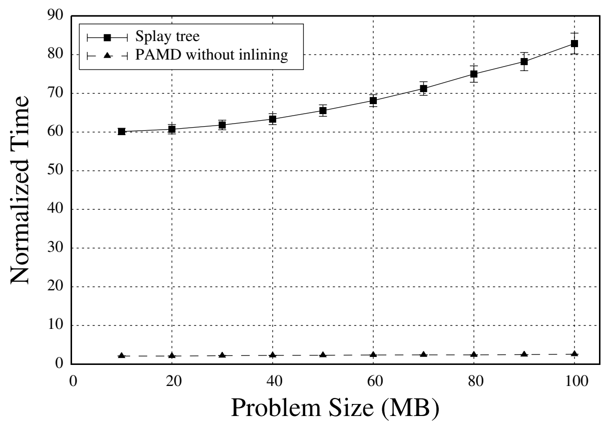
6. Related Work

Many security hardening and debugging tools need the ability to find metadata attached to a pointer's referent memory object. Applications include memory safety [10, 11, 18], data race detection [8, 29], and dynamic information flow enforcement [21]. Many methods use shadow tables (the equivalent of the baggy bounds table we employ) with metadata stored on a per-slot basis. Examples include Address Sanitizer [30], Baggy Bounds Checking [6], WIT [5], SoftBound [26], and SharC [8]. Shadow tables replicate metadata by recording it for each slot occupied by a memory object; as the size of memory objects and the metadata grows, so does the shadow table. A common challenge in such systems is to keep each entry in the shadow table small; in some cases, functionality is sacrificed to make this possible [8]. In contrast, BBAC [14] and PAMD keep the shadow table small and leverage the padding added by increasing the allocation size to a power of two to store *one* copy of the metadata for each memory object.

Memory safety is a primary consumer of metadata. Early memory safety systems such as those by Jones and Kelly [18] and Ruwase and Lam [28] used a single splay tree to record memory object bounds information. Dhurjati and Adve [12] used points-to analysis to split a single splay tree into multiple splay trees to speed up metadata lookup; the original Secure Virtual Architecture [11] used this approach to efficiently enforce memory safety for operating system kernel code. Other memory safety systems forwent metadata lookup by extending the pointer representation [17] or segregating the heap [13]. The former suffers from compatibility issues with external library code while the latter requires sophisticated inter-procedural transformation and special-



(a) Bzip2 Allocation Size and Object Count vs Problem Size



(b) Bzip2 Execution Time vs Problem Size

Figure 13. Bzip2 Scalability Results

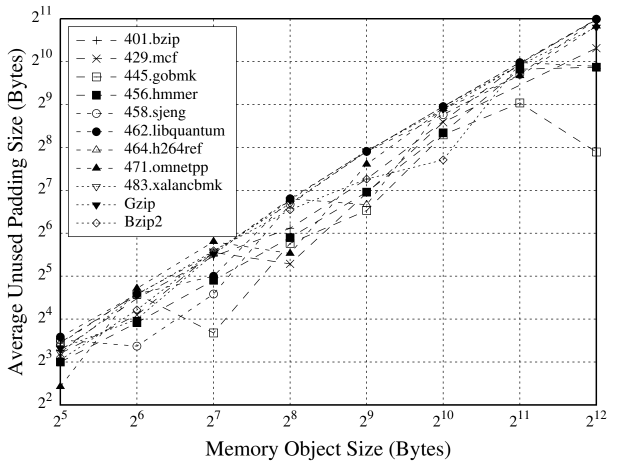


Figure 14. Average Unused Padding Size

ized memory allocators. Tools such as `libcrunch` [20] (which uses `liballocs` [19]) dynamically detect bad type casts instead of checking memory access and pointer arithmetic operations. These tools catch different types of errors than traditional memory safety tools and are faster than earlier tools because the operations they instrument are executed less frequently. BBAC [14] is essentially PAMD used for memory safety with constant-sized metadata; our PAMD implementation is more robust and efficient, and we evaluate PAMD's scalability and sources of overhead.

Comparing PAMD's performance with previous tools is challenging; performance depends heavily on which operations are checked, which optimizations are employed, and how the run-time checks are designed and implemented. For example, WIT [5] and Address Sanitizer [30] are faster than PAMD (10% and 73% overhead on SPECint, respectively, versus 2.5× for PAMD) because they do not check loads from memory and pointer arithmetic, respectively. Consequently, PAMD can detect errors that WIT and Address Sanitizer cannot. There are also engineering tradeoffs. Dhurjati and Adve [12] reduced splay tree overhead to 12% on Olden by utilizing multiple splay trees and applying several other optimizations; PAMD overhead is 75% on Olden. The Dhurjati and Adve optimization [12] comes at the cost of building and maintaining a sophisticated inter-procedural points-to analysis named Data Structure Analysis (DSA) [24] (it may also fail to scale; even with multiple splay trees, splay trees will grow larger as input size increases). PAMD, on the other hand, achieves its constant-time performance without the aid of sophisticated static analysis, and our experiments demonstrate its scalability.

7. Future Work

Constant-time metadata lookup enables the design and implementation of per-memory object policies. For example, run-time checks could use the metadata to ensure that loads and stores access memory objects in the correct points-to set, making points-to analysis sound for all program executions. The only existing approach of which we know can only check for compliance with a unification-based points-to analysis [13]. Using our method, memory objects could be labeled with the points-to sets to which they belong; run-time checks before loads and stores could ensure that the referent memory object of the pointer used in the load or store is in one of the correct points-to sets. Such an approach could check for violations of subset-based points-to analysis algorithms [7].

Our results point to further research on implementing optimizations. We will explore inter-procedural optimizations that eliminate unneeded object registration and deregistration to reduce overhead. Some tools using metadata can elide the registration and deregistration of memory objects if a run-time check will never need the metadata. This can happen, for example, if all the run-time checks are in the same function in which the memory object is allocated and no pointer to the memory object escapes to memory. We can also study inter-procedural optimizations that eliminate redundant checks across function boundaries.

8. Conclusions

We presented PAMD, a method of attaching metadata of arbitrary size to memory objects and a method of locating that metadata in constant time. We evaluated the performance of PAMD within the SAFECode compiler which performs memory safety checks on memory access and pointer arithmetic operations. Our results show that, for dynamic memory safety error checks, our method induces little overhead above BBC [6] which trades precision for performance by using a simple shadow table. We also demonstrated that PAMD scales well when instrumented programs allocate more memory objects in response to larger inputs, showing relatively constant overheads as input sizes increase.

References

[1] Database speed comparison. https://www.sqlite.org/speed.html.

[2] The python benchmark suite. https://github.com/python/performance.

[3] SAFECode. http://sva.cs.illinois.edu.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security*, 13:4:1–4:40, November 2009.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[6] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[7] L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[8] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded c. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 149–158, New York, NY, USA, 2008. ACM.

[9] D. P. Bovet and M. Cesati. *Understanding the LINUX Kernel*. O'Reilly, Sebastopol, CA, 3^{rd} edition, 2006.

[10] J. Criswell, N. Geoffray, and V. Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[11] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, Stevenson, WA, USA, October 2007.

[12] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *International Conference on Software Engineering*, Shanghai, China, May 2006.

[13] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[14] B. Ding, Y. He, Y. Wu, A. Miller, and J. Criswell. Baggy bounds with accurate checking. In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012.

[15] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, Sept. 2006.

[16] A. Jaeger. Porting to 64-bit gnu/linux systems. In *Proceedings of the GCC Developers Summit*, pages 107–121, 2003.

[17] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, 2002.

[18] 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*, pages 13–26, 1997.

[19] S. Kell. Towards a dynamic object model within unix processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 224–239, New York, NY, USA, 2015. ACM.

[20] S. Kell. Dynamically diagnosing type errors in unsafe code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 800–819, New York, NY, USA, 2016. ACM.

[21] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the Conference on Code Generation and Optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.

[23] C. Lattner et al. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html.

[24] C. Lattner, A. D. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289, San Diego, CA, USA, June 2007.

[25] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, second edition, 2015.

[26] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[27] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Trans. on Prog. Lang. and Sys.*, 17(2), Mar. 1995.

[28] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, San Diego, CA, USA, 2004.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.

[30] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX.

[31] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. In *Proc. of the ACM Symp. on Theory of computing*, 1983.