Safe Manual Memory Management

David Gay Rob Ennals Eric Brewer
Intel Research Berkeley
{david.e.gay,robert.ennals,eric.a.brewer}@intel.com

Abstract

We present HeapSafe, a tool that uses reference counting to dynamically verify the soundness of manual memory management of C programs. HeapSafe relies on a simple extension to the usual malloc/free memory management API: *delayed free scopes* during which otherwise dangling references can exist. Porting programs for use with HeapSafe typically requires little effort (on average 0.6% of lines change), adds an average 11% time overhead (84% in the worst case), and increases space usage by an average of 13%. These results are based on porting over half a million lines of C code, including perl where we found six previously unknown bugs.

Many existing C programs continue to use unchecked manual memory management. One reason is that programmers fear that moving to garbage collection is too big a risk. We believe that HeapSafe is a practical way to provide safe memory management for such programs. Since HeapSafe checks existing memory management rather than changing it, programmers need not worry that HeapSafe will introduce new bugs; and, since HeapSafe does not manage memory itself, programmers can choose to deploy their programs without HeapSafe if performance is critical (a simple header file allows HeapSafe programs to compile and run with a regular C compiler). In contrast, we found that garbage collection, although faster, had much higher space overhead, and occasionally caused a space-usage explosion that made the program unusable.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic Storage Management

General Terms Languages

Keywords Memory Management, Reference Counting, C, Safety

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISMM '07 October 21–22, Montréal, Québec, Canada.
Copyright © 2007 ACM [to be supplied]. . . $5.00

1. Introduction

Memory management bugs are a significant cause of software failures and vulnerabilities in C programs. Although some C programs have addressed this problem by using garbage collection [7], a large proportion of C programs continue to use unchecked manual memory management. Anecdotal evidence suggests that concerns about performance and software engineering issues are the two main factors that dissuade C programmers from using garbage collection. On the performance side, programmers believe that garbage collection's performance overhead is too high for performance-critical programs, its memory overhead is too high for space-critical programs, its scaling is not good enough for concurrent programs, and its pauses are too long and too unpredictable for real-time programs. On the software engineering side, using garbage collection has several drawbacks. First, you abdicate memory management to a complex runtime system that you do not control. Second, your program becomes "locked in" to garbage collection and will not work correctly without it. Finally, applying garbage collection to existing code has the potential to change behavior and introduce memory leaks.

Although performance issues have been largely solved, or at least alleviated, in recent garbage-collector designs [32, 33, 7], the software engineering issues still present a significant barrier to adopting garbage collection in legacy C code. This is particularly the case for systems, media, and real-time code, where programmers often believe they need fine control over performance; and for mission critical software, for which far-reaching changes are considered too risky.

An alternative to garbage collection is to check the safety of a program's existing memory management. We have designed and implemented HeapSafe, a C-to-C compiler and runtime system that uses automatic reference counting to dynamically check that there are no references to an object at the point at which it is freed. HeapSafe requires some small source code changes, and adds some (reasonable) time and space overhead. Since HeapSafe checks, rather than replaces, a program's existing memory management, programmers need not worry that it will change the behavior of their program (but see Section 2.4 for a discussion of corner cases in which behavior might change) or that their programs will become dependent on it. Similarly, since HeapSafe does not

```
void free_loop(struct loop *start) {
    struct loop *next, *cur = start;
    delayed_free_start(); /* API change */
    do {
        next = cur->next;
        free(cur);
        cur = next;
    } while(cur != start);
    delayed_free_end(); /* API change */
}
```

Figure 1. While freeing this cyclic list, the `next` pointer from the node before `start` dangles.

introduce unpredictable pauses, and has moderate performance and memory overheads, it is practical to use it for deployed code, as an alternative to a garbage collector.

The key to our approach is a small extension to the standard "allocate and free" manual memory management API. This change is the introduction of *delayed free scopes* within which deallocations are delayed and references to deallocated objects can exist. Dynamically checking calls to `free` has historically been seen as impractical, since real code often contains benign references to deallocated objects that are in fact never dereferenced. The insight behind delayed free scopes is that these benign references are typically short lived: they are found in local variables that will soon die, global variables that will soon be overwritten, or in other parts of the same data structure that will soon be freed. Using delayed free scopes, the programmer tells the system when to check that the deallocations are safe, i.e., that no references remain to the deallocated objects.

For instance, in the `free_loop` function of Figure 1 that frees a circular list, the dangling reference to the first list element goes away once the last element of the circular list is freed. The delayed free scope, indicated by the calls to `delayed_free_start` and `delayed_free_end` tells the system to delay the deallocations and deallocation checks until the end of `free_loop`.

HeapSafe has three significant limitations. First, it is not fully sound (i.e., it can miss or misreport bad frees), mostly because of limitations of C itself, such as the lack of array bounds checks (Section 2.6) — a safer base language would help here. Second, HeapSafe does not yet support multi-threaded code. We have performed some preliminary work in this area by applying HeapSafe to the Linux kernel [1], but have not yet resolved all issues to our satisfaction (Section 3.4). Finally, HeapSafe does not detect memory leaks, as it only checks the soundness of existing frees. HeapSafe could be extended with a partial leak detector, based on finding unfreed objects with a zero reference count, but this would fail to detect unreachable cyclic structures.

We have applied our HeapSafe implementation (Section 3) to a 558k line codebase (Section 4), including all malloc/free-based C programs from SPEC2000 and SPEC2006 [30].[1] Porting code to HeapSafe required changes to between 0% and 2% of source code lines, with a mean of 0.6%. HeapSafe's performance is reasonable — on average 11% slower and using 13% more memory, though the worst case slowdown is 84% on perl.

HeapSafe is intended to be used as an "always on" dynamic correctness checker, rather than a "debug only" bug finder. The purpose of these experiments was thus to show that HeapSafe can be easily applied to existing C programs, rather than to see how many bugs we could find. We ran these programs on well-tested inputs — to seriously search for bugs, we would subject these programs to a much wider variety of inputs. Nevertheless, we found six new bugs in perl as a result of trying to understand and check perl's memory management.

We also compared (Section 4) HeapSafe to the Boehm-Weiser conservative garbage collector, and found that HeapSafe typically has higher time overhead (11% vs 5%), but lower space overhead (13% vs 85%). Additionally, the variance of space overhead for garbage collection was far higher, with some programs above 200% and others failing due to running out of memory. Furthermore, even though our goal is to lower HeapSafe's overhead sufficiently that it can always be used, a programmer can always chose to run a HeapSafe program without HeapSafe, using a conventional C compiler and library. Conversely, a program that runs fine with a conservative garbage collector may leak memory when it is not present. Section 5 compares HeapSafe to related work in more detail.

In summary, this paper makes two significant contributions. First, we introduce the idea of *delayed frees*, which allow programmers to specify points at which deallocation can effectively be checked. Delayed frees are presented in HeapSafe using a simple API: *delayed free scopes*. Second, we show that by combining delayed free scopes with automatic reference counting, it is practical to check the correctness of explicit memory management in existing code, even for large complex programs. We believe this result will also apply to other languages with explicit memory management.

2. HeapSafe

HeapSafe uses reference counting and delayed free scopes to check the correctness of existing malloc/free calls in single-threaded C programs. The programmer typically needs to make a few small modifications to their program to avoid HeapSafe incorrectly reporting bad frees. Since the intention of HeapSafe is to check memory management, rather than change it, these modifications do not change the observable behavior of the program (modulo the issues discussed in Section 2.4).

These modifications fall into two categories. First (Section 2.1), programmers must ensure that objects are no

[1] Except for `libquantum`, which uses complex numbers which are unsupported by the CIL framework [23] on which HeapSafe is based.

(a) Original code:

```
void free_exp(struct exp *e) {
    free(e->type);
    free(e);
}
```

(b) With explicit nulling:

```
void free_exp(struct exp *e) {
    struct type *tmp = e->type;
    e->type = NULL;
    free(tmp);
    free(e);
}
```

(c) With ZFREE:

```
void free_exp(struct exp *e) {
    ZFREE(e->type);
    free(e);
}
```

Figure 2. The child object is freed before the parent: `e->type` is a short-lived dangling reference.

longer referenced when their deallocation is checked. This can be ensured by adding delayed free scopes, or by adding explicit code to null-out dangling references (e.g., in global variables). Second (Section 2.2), to ensure the accuracy of reference counting, most programs require some changes to help HeapSafe locate pointers and pointer writes (e.g., by replacing calls to `memcpy`, `memset` and `memmove` by type-aware versions). Altogether, these changes typically affect less than 1% of source code.

HeapSafe programs are simply C programs written with an extended memory allocation API. As a result, regular C programs can be compiled with HeapSafe (Section 2.3), HeapSafe programs can be compiled by a regular C compiler with a small header file (Section 2.4), and code compiled by HeapSafe and a regular C compiler can be linked together, e.g., when using the standard C library in a HeapSafe program (Section 2.5). We end this section with a summary of all the issues that can cause HeapSafe to miss or misreport bad frees (Section 2.6).

2.1 Checking Frees

HeapSafe maintains an integer reference count for each `malloc`-allocated object, recording the number of pointers that refer to that object. When the program calls `free` on an object, HeapSafe will verify that the reference count is zero. Unlike conventional reference counting, HeapSafe will not automatically free objects when their reference count reaches zero. It is thus necessary for the programmer to free all objects explicitly, just as they would if HeapSafe was not being used.

Applying HeapSafe to existing C code typically reports a number of bad frees, even if the code is correct, because C programs often keep a pointer to an object after the object is freed. Such *dangling references* are usually benign, and exist purely because it is more convenient to free the object before removing references to it. For example, a program will often free an object before freeing a containing object (Figure 2a).

One can often avoid short-lived dangling references by simply zeroing the remaining reference to the object being freed (Figure 2b). Since such conversions are awkward for the programmer and provide an opportunity to introduce errors, HeapSafe defines a macro, `ZFREE` that automatically zeroes the reference (Figure 2c).

It is usually, though not always, safe to replace any call to `free` with a call to `ZFREE`. Dereferencing a freed pointer is undefined behavior and so problems can only arise if a program wishes to compare freed pointers, or test a freed pointer for NULL. Perl contains uses of both these idioms...

HeapSafe uses a simple liveness analysis to determine when a local variable is dead. If a variable does not have its address taken, and is not used after the call to `free` then it need not be zeroed.[2]

2.1.1 Delayed Frees

Zeroing references is not always practical. For instance, when freeing a circular list (Figure 1), the programmer would have to add additional code to make the list non-circular before starting to free the list. However, as we discussed in the introduction, the dangling reference from the last element of the circular list is short-lived: it no longer exists at the end of `free_loop`.

We can exploit this kind of property by delaying a free until all dangling references to the freed object have gone away. More specifically, we want to delay the actual execution of that free operation and the corresponding reference count check until such time as any transient references have gone away.

Our initial approach to delaying frees was to collect objects into a group, and free the group only after all dangling references had been removed. While this works, we found that it often requires many modifications to existing C code, increases the likelihood of errors and reduces readability.

Instead, HeapSafe provides *delayed free scopes*, an API for delayed frees based on a system of nested scopes. The program calls `delayed_free_start` to start a *delayed free scope* and calls `delayed_free_end` to end it. Within a delayed free scope, all calls to `free`, and their checks, are delayed until the end of the scope, at which point all dangling references should have disappeared. If scopes are nested then the inner scope is absorbed into the outer scope; frees will be delayed until the end of the outermost scope. Frees that occur outside any scope are performed and checked immediately. In the case of Figure 1, we simply wrapped the body of the `free_loop` function in a delayed free scope.

[2] Without this analysis, most frees outside a delayed free scope would be bad.

Delayed scopes are only practical if the scopes are kept relatively small. In the extreme case, one could wrap the entire program with a delayed free scope, delaying all frees until the end of the program and essentially causing all memory to be leaked. Fortunately, we have found that, in practice, we can keep delayed free scopes relatively small, typically causing a negligible impact on the memory footprint of a program (Section 4.3).

2.1.2 Disabling Reference Counting

In some unusual cases, programs contain frees that are safe, but where the dangling references cannot be easily found. For example, perl caches methods with an associated generation number. If this generation number is different from the global generation number, then the entry isn't used. Operations that might free methods increment the global generation number, rather than trying to remove all dangling cached methods. With HeapSafe, all such frees would be reported as bad frees. Unsurprisingly, perl does not use its own internal reference counting for these cached methods.

As a workaround, HeapSafe programs can declare pointers with the `norefcount` qualifier to disable all reference counting operations. This qualifier is also useful when interacting with code not compiled with HeapSafe, as we discuss further in Section 2.5.

2.2 Finding Pointers

When an object o is freed or overwritten, HeapSafe needs to know the location of all pointers in o so that it can adjust reference counts for referenced objects. HeapSafe also needs a conservative estimation of the locations containing pointers when objects are allocated, so that all pointers can be initialized to `NULL`.

The correctness of this type information is not essential for running HeapSafe programs, but incorrect type information can lead to inaccurate bad free reports. For instance, misidentification of an integer with the same bit pattern as a pointer will lead to an incorrect reference count. This can cause incorrect bad free reports and, if bad frees are configured to leak (Section 2.3), memory leaks.

To ensure that all pointers are initialized to NULL, HeapSafe initializes all local pointer variables to NULL,[3] HeapSafe's version of `malloc` zeroes allocated objects, and `realloc` zeroes out the new part of a reallocated object. For increased performance (zeroing can have a significant performance impact in allocation-intensive programs), HeapSafe provides typed allocation functions which only zero out pointers.

HeapSafe locates pointers by making the following assumptions about C programs: objects and parts of objects are always written with the correct type, the type passed to `free` represents the exact type of the object being freed, and unions only have their "type" modified when all possible pointer fields are NULL. Additionally, HeapSafe assumes that a freed pointer points to an array of elements extending until the end of the allocated heap object, or, if the type is a struct ending in an open array, that the final array field extends to the end of the allocated heap object. HeapSafe locates pointers within objects based on their declared type when possible, but requires that the programmer write an *adjust function* when not. The role of an adjust function is to find and adjust the reference count of all pointers in a given object.

Formally, adjust functions have the following signature:

```
void adjust_X(void *object, int inc, size_t s);
```

Such a function should find all pointers in `object` and adjust the reference counts of the objects they point to by `inc`, taking into account the fact that `object` has type `X` and size `s`. The HeapSafe library contains utility macros to make adjusting reference counts and looping over all objects simple.

Common cases requiring an adjust function include unions containing pointer fields, object-oriented-style code (using structs A and B, where the fields of A are a prefix of those of B, and pointers to the structs are cast to each other), and cases where pointers are encoded as integers.

Because C is not a type-safe language, HeapSafe's assumptions may be incorrect. HeapSafe gives warnings for various constructions that are likely to be problematic. First, HeapSafe warns whenever `free` is passed a `void*` or `char*` pointer (for historical reasons, many C programs cast pointers to `void*` or `char*` before freeing them). Second, it reports whenever `memcpy`, `memset` and friends are used to copy types containing pointers — such calls must be replaced in HeapSafe by type-aware copy and clear routines. Finally, HeapSafe warns the user in most cases where adjust functions are needed.

We believe that combining HeapSafe with a tool designed to ensure type safety for C, such as Deputy [8], CCured [22], or Automatic Pool Allocation [10] would reduce the burden of these changes on the HeapSafe programmer.

2.3 Dealing with Bad Frees

The user decides what HeapSafe should do on bad frees. The default behavior is to free the object regardless, but log the error. This ensures that behavior will be the same as without HeapSafe. The programmer can also opt to have HeapSafe not free objects whose check fails, which is safer, but risks introducing space leaks[4].

With HeapSafe's default behavior, existing C code runs unchanged under HeapSafe, even if reference counts are wrong. This provides a good first step when porting C code to HeapSafe, as we discuss further in Section 4.2. Furthermore, this remains true during the porting process: a par-

[3] C already ensures that all global variables are so initialized.

[4] We have considered providing the option of a periodic scan that frees any such objects if their reference count has subsequently reached zero, but have not yet implemented this.

tially or incorrectly ported program still functions correctly. Missing scopes, ZFREEs, or adjust functions (Section 2.2), can only cause harmless frees to be logged as bad frees or already-bad frees to be missed.

2.4 Erasure Semantics

One of the goals of HeapSafe is to allow programs written for HeapSafe to be compiled and run with any C compiler. All HeapSafe files must `#include <heapsafe.h>`. When compiled with a regular C compiler this header file defines HeapSafe's API in terms of regular C operations. The only differences between the resulting program and one compiled with HeapSafe are:

- **HeapSafe detects bad frees:** HeapSafe will log a bad free if the reference count check fails on a call to `free`. The programmer can request arbitrary behavior check failure (Section 2.3).
- **Delayed frees hide errors:** If a program frees an object within a delayed free scope, and then erroneously accesses the object before the delayed scope has ended then HeapSafe will hide this error. If the program was run without HeapSafe then it is possible that the program would have accessed a newly allocated object. Programmers can avoid this issue by using a library that provides (unchecked) delayed frees for their normal C compiler.
- **Performance differences:** Programs compiled with HeapSafe have increases in the memory usage due to delayed frees and extra space for reference counts (Section 4.3), and decreases in performance, due to the overhead of reference count operations (Section 4.4).

2.5 Mixing HeapSafe and C Code

HeapSafe can only track references within code that it compiles. If a pointer is passed to a foreign library that was not compiled with HeapSafe then HeapSafe will not be aware of any references that are held by that library.

Unlike conventional reference counting, it is acceptable for a library to hold its own private references to a reference counted object. With conventional reference counting, since objects are freed when their reference count reaches zero, un-tracked references can cause referenced objects to be freed. With HeapSafe, un-counted references within a library can cause false negatives (missing bad free reports) but not false positives (extra bad free reports).

However, reference counts may become incorrect if the reference-counted program and the non-reference counting library overwrite each other's pointers (e.g., the program stores a pointer, incrementing its count, and the library overwrites it, not decrementing the count). The same problem arises with deallocation: if the program frees an object containing library-written pointers, HeapSafe will decrement the reference counts for the pointers written by the library.

One workaround for these problems is to declare the types of pointers written by both the library and the main program with the `norefcount` annotation, to disable reference counting. In the current version of HeapSafe it is up to the programmer to identify such types, and verify the correctness of the resulting memory management.

At least for the standard C library, problematic writes by library code are rare except for the previously mentioned `memcpy`, `memset` and `memmmove` functions. None of our programs had problems with reference counting due to use of any other C library functions.

2.6 Soundness

The soundness of HeapSafe is limited by several factors. First, HeapSafe uses 8-bit reference counts, so bad frees of objects with $k \equiv 0 \mod 256$ references references are missed (Section 3.1.1) — we believe this is very unlikely in non-malicious code.

Second, HeapSafe does not check for type-safety issues such as array bound overflows, walking pointers past the end of arrays, incorrect use of unions and casts, etc, which can cause or hide memory safety problems. Third, HeapSafe relies on the programmer to provide precise information on the location of all pointers and pointer writes in cases where this information is normally unavailable in C (Section 2.2). Combining HeapSafe with a tool designed to ensure type safety for C [8, 22, 10] would resolve these last two issues.

Finally, programmers can disable reference counting, on a type-by-type (Section 2.1.2) or file-by-file (Section 2.5) basis. It is then up to them to ensure that this trusted code does not cause memory safety violations.

3. Implementation

HeapSafe tracks object reference counts by associating a 1-byte reference count with each 8-byte block of memory (Section 3.1). These reference counts are updated whenever pointers are written into the heap, or whenever an object containing pointers is freed (Section 3.2). A variant of deferred reference counting is used to manage stack pointers. The current version of HeapSafe is designed for single-threaded code, but can be used for multi-threaded code with some significant caveats (Section 3.4).

At present, HeapSafe only checks the correctness of calls to `free`. It does not check that there are no references to stack allocated data when the stack frame is popped. We believe it should be relatively easy to add such a feature to HeapSafe, but leave that for future work.

HeapSafe is implemented as a C-to-C translator, built over the CIL [23] infrastructure. HeapSafe's output is compiled by gcc. HeapSafe's runtime library is based on Doug Lea's malloc/free library v2.8.3.[5]

3.1 Storing Counts

HeapSafe maintains a reference count table that contains a reference count for every 8-byte block of memory (Figure 3).

[5] `http://gee.cs.oswego.edu/dl/html/malloc.html`

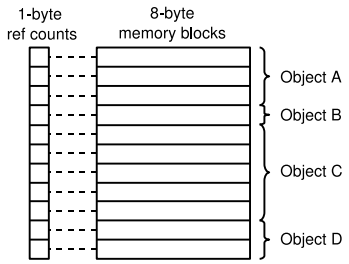

Figure 3. The Reference Count Table stores a 1-byte reference count for every 8-byte block in the heap

Placing the reference counts in a separate table rather than inside the object itself allows HeapSafe to avoid changing the data representation, and allows HeapSafe to cope with pointers to the interior of objects, from which it may be hard to find the start of the object. When an object is freed HeapSafe sums the reference counts for all the 8-byte blocks covered by the object. If the counts sum to zero then there are no references to the object and it can be safely freed.

HeapSafe checks the block reference count sum (rather than simply checking that all blocks have a zero reference count) to avoid the need for reference counting operations on pointer arithmetic operations. For instance, the sequence

```
p = malloc(16); p += 8; p = NULL;
```

will leave the reference counts of the first two blocks of the allocated object at 1 and -1 respectively, but the sum is still zero. Note that this requires that pointer arithmetic keep pointers within the same object, as required by the C standard. Type safety tools for C, such as Deputy [8] already enforce this restriction. The C standard does allows a pointer to an object to point to the byte after the end of the object, so this byte must also be contained in the blocks assigned to that object. Fortunately, in the case of the Doug Lea allocator, each allocated object already has a 1-word header, placed at the end of the previous 8-byte block, so this byte is already reserved.

The choice of 8-byte blocks is a compromise. Larger blocks reduce the amount of space needed for the reference count table, while smaller blocks reduce the amount of space wasted by requiring all objects actual size to be a multiple of the block size. We chose 8-bytes partly because this is the size-granularity used by the Doug Lea allocator, which HeapSafe is based on. Similarly, the choice of 1-byte reference counts is a compromise. Smaller reference counts save space while larger reference counts reduce the chance of overflow (Section 3.1.1).

To simplify the process of finding the reference count for a block, HeapSafe allocates a single array with an entry for every block in the entire address space. For a 32-bit address space with 1-byte reference counts this is a 512MB table. Currently, HeapSafe relies on Linux's lazy page allocation to allocate physical pages, and thus only pays a space cost proportional to allocated object size (plus an overhead for pointers to stack, code and globals). On 64-bit operating systems, or operating systems without lazy page allocation, the HeapSafe library should instead explicitly map the necessary reference counting pages when first using a new section of the address space.

Since the table is direct mapped and we do not check for overflow when changing a reference (see below), the reference adjustment macro is very simple:

```
#define REFCOUNT_ADJUST(x, by) \
  if (x) __rcs[(intptr_t)(x) >> BLOCKSHIFT] += by
```

The test that `x` is non-null is not necessary, but improves performance.

3.1.1 Overflowed Counts

HeapSafe does not check for overflow of its reference counts. This is acceptable as HeapSafe is being used to check the correctness of non-malicious code with explicit frees: at free-time, an unreferenced object's reference count will still be zero even if overflows occurred, and for referenced objects it is unlikely that the number of references will be an exact multiple of 256 when a bad free occurs. Even if the number of references is a multiple of 256, the only consequence will be a failure to log a bad free. Conventional reference counting cannot omit this check for overflow, since allowing a count to roll over to zero could cause referenced objects to be freed.

To guarantee memory safety, HeapSafe could perform an explicit (signed) overflow check on every reference count operation, triggering an appropriate recovery operation. This would however increase runtime overhead.

3.2 Tracking Counts

HeapSafe's C-to-C translator augments the original program with code that keeps track of the number of references to each allocated heap object. To do this, it must track all pointer writes. HeapSafe uses a variation on deferred reference counting [9] to track local variables (which contribute a significant majority of pointer writes), and updates reference counts immediately for writes to global variables and the heap.

For pointer writes, the object pointed to has its reference count incremented and the overwritten pointer has its object's reference count decremented, using the `REFCOUNT_ADJUST` macro above. For writes of objects containing pointers, HeapSafe calls the adjust function for the written type before the write to remove references from the old value, and calls it after the write to add references from the new value.

When an object is freed, HeapSafe calls the object type's adjust function to remove references from the freed object to the rest of the heap. When freeing a delayed scope, all objects have their references removed before any object's refer-

ence count is checked. This is necessary to avoid erroneously reporting bad frees.

HeapSafe uses deferred reference counting [9] to avoid reference counting local variables. Conventional deferred reference counting places an object in a Zero Count Table (ZCT) when its reference count reaches zero. To actually free objects, the stack is scanned and referenced objects are removed from the ZCT and the other objects are freed. HeapSafe's deferred reference counting is similar: local variables are not reference counted, and at free time the stack is scanned to check that there are no references to the object to be freed. The difference is that HeapSafe's free checks are explicitly requested by the programmer, occurring at each free outside a scope[6]and at the end of delayed free scopes.

Deferred reference counting increases the pause at free time, but this overhead is small and predictable, as it is a function of the number of live variables on the call stack.

Our current implementation of HeapSafe generates portable ANSI-C, and so is not able to do the low-level stack walk required for truly efficient deferred reference counting. Instead, it maintains a separate *shadow stack* in which all local pointer variables are stored, and checks this stack for references. HeapSafe performs some simple optimisations (removing adjacent pop/push pairs, hoisting push/pop out of loops) to increase performance. However, maintaining this shadow stack can still have significant overhead, as discussed in Section 4.4.1. Previous work has shown that, if sufficient care is taken with implementation, deferred reference counting can give very good performance [19].

To reduce the shadow stack overhead, HeapSafe allows programmers to annotate functions with `nofree` if the function never calls `free` directly or indirectly. Functions annotated with `nofree` can only call other functions annotated with `nofree`. HeapSafe will not bother saving variables on the shadow stack around calls to `nofree` functions. This is a temporary solution, but yields worthwhile performance improvements when applied to inner loops.

3.2.1 Setjmp/Longjmp

A significant advantage of deferred reference counting is that it makes it practical to support `setjmp` and `longjmp` in a C-to-C translation system. In `setjmp`, we remember the position on our shadow stack. In `longjmp`, we restore the position of the shadow stack before returning to the `setjmp` point.

Additionally, `setjmp` saves, and `longjmp` restores the current delayed free scope nesting depth. If the old depth was non-zero, and the new depth is zero, the delayed frees are performed at `longjmp` time, after the shadow stack pointer is restored.

Local arrays, structures, and variables whose address is taken are treated in hybrid fashion. If a pointer is written to such a stack object then it is reference counted immediately (since other code may be treating it as a heap object); however all such stack objects are also recorded in an additional shadow stack, allowing `longjmp` to decrement the reference counts of any referenced objects when the stack objects go out of scope.

3.3 Delayed Free Scopes

The implementation of delayed free scopes is a straightforward chunky list of pointer and type pairs of the objects to be freed. The first element of the chunky list is statically allocated, subsequent elements are dynamically allocated as needed and freed when the delayed free scope ends.

If no memory is available for the chunky list, the system falls back to doing unchecked, immediate frees.

3.4 Multi-threading

There are several problems with HeapSafe's implementation in a multi-threaded environment:

- **Reference counts are not updated atomically.** If two threads try to manipulate the same reference count at the same time then only one update may be visible.
- **Pointer writes are not atomic.** When overwriting a pointer HeapSafe needs to know what pointer is being overwritten, so that it can decrement the object. Since the update is not atomic, another thread could overwrite a pointer between HeapSafe reading the old value and writing the new value.
- **Summing reference counts is not atomic.** When an object is freed, HeapSafe checks that the reference counts for all blocks in the object sum to zero, however this summation is not done atomically (Section 3.1). It is possible that another thread might release and create references to an object while HeapSafe is walking over the reference count array, causing HeapSafe to conclude incorrectly that the object is not referenced.
- **Other threads may have deferred references to an object.** When freeing an object, deferred references from other threads need to be taken into account.

Under the assumption that pointer writes are not subject to races,[7]we can address the first three issues with two small modifications to HeapSafe: increment reference counts atomically, and perform reference count increments before decrements. We have successfully used this modified version of HeapSafe to check memory allocation in the Linux kernel [1][8], but have not addressed the issues of performance and deferred references from other threads. Previous work on multi-threaded reference counting [19] leads us to believe that these problems can be solved.

[6] Batching unscoped frees did not improve performance because stack scanning can be optimised for a single object.

[8] In that paper, HeapSafe is referred to under the name CCount.

[8] A static checker tool such as as RacerX [12], or a dynamic race detector such as Eraser [28], could be used to check for such races.

	Source Code							Max Heap Usage (MB)			Runtime (mins)			
Benchmark	kLOC	Changes	S	A	F	N	O	Orig.	HeapS.	GC	Orig.	HeapS.	GC	alloc/s
SPEC2000:														
crafty	21.2	0.25%	2	0	0	47	3	0.88	12.5%	42.6%	1.2	1.8%	-0.0%	0.55
gzip (5)	8.6	0.42%	0	18	14	0	4	189	12.5%	39.4%	0.38	2.7%	2.1%	873
parser	11.4	2.1%	52	8	64	51	60	12.9	12.8%	116%	2.3	33.2%	4.4%	5.73M
twolf	20.5	0.37%	14	0	8	0	54	3.4	22.8%	8.5%	2.4	11.7%	4.2%	3.96k
art (2)	1.3	0.00%	0	0	0	0	0	3.5	12.5%	17.7%	0.82	0.30%	2.9%	618
equake	1.5	0.00%	0	0	0	0	0	43.4	12.5%	8.9%	1.4	-2.8%	9.1%	16.3k
ammp	13.5	0.68%	0	0	20	47	25	13.9	12.5%	161%	3.7	3.9%	25.9%	174
mesa	61.7	0.26%	46	0	14	66	35	21.4	12.5%	8.6%	1.8	-2.4%	0.50%	0.57
SPEC2006:														
perl (3)	168	1.0%	347	350	63	96	853	290	12.8%	85.2%	3.3	84.1%	-0.5%	350k
bzip2 (6)	8.3	0.22%	4	0	5	0	9	414	12.5%	24.2%	2.3	4.4%	-0.0%	0.20
mcf	2.7	1.0%	8	0	0	5	14	879	12.5%	0.76%	10.8	8.0%	-0.8%	0.01
gobmk (5)	61.2	0.18%	12	0	8	74	14	16.6	12.5%	208%	2.5	19.3%	0.37%	788
hmmer (2)	36.0	1.1%	14	0	364	0	22	8.6	12.5%	286%	10.0	2.0%	1.0%	2.02k
h264ref (3)	51.6	0.78%	28	50	108	115	99	31.4	12.8%	352%	4.3	14.5%	1.5%	203
milc	15.0	0.45%	4	0	0	42	21	702	12.5%	Failed	20.3	6.7%	Failed	5.4
lbm	1.2	0.17%	2	0	0	0	0	429	12.5%	4.0%	21.7	-0.1%	-2.4%	$<$0.01
sphinx3	25.1	0.95%	54	14	154	0	16	41.5	12.2%	Failed	18.8	5.0%	Failed	23.1k
Misc:														
cfrac	4.2	0.36%	4	0	1	2	8	0.46	13.9%	150%	0.18	20.2%	4.7%	5.19M
grobner	13.5	0.57%	0	0	26	32	19	0.36	12.5%	227%	0.19	21.6%	28.8%	5.83M
tile	4.9	0.84%	0	0	35	2	4	0.96	12.5%	22.0%	0.21	0.05%	0.76%	2.20k
espresso	15.3	0.96%	14	0	6	36	90	0.38	12.5%	389%	0.08	21.4%	25.3%	890k
boxed-sim	11.6	0.14%	2	0	1	13	0	0.36	12.7%	100%	0.14	13.6%	-0.7%	30.1k
Total	558		607	440	891	628	1350							
Geo Mean		0.58%							13.0%	84.7%		11.1%	5.0%	

Changes are: S = scopes, A = adjust functions, F = changes to free functions, N = nofree annotations, O = other changes (typically nulling).
Changes are measured in lines of code. `gobmk` contains 197kloc of C, however 136kloc of this is encoded data.
A benchmark run on $n > 1$ inputs is marked with (n). Runtime and heap usage figures are geometric averages over all inputs.

Figure 4. Benchmark Statistics

4. Evaluation

To demonstrate the practicality of HeapSafe, we ported 22 programs totaling 558k lines of C code. Most programs ported in a few hours, with typically less than 1% of lines needing to be annotated (Section 4.2). HeapSafe increases code size by an average of 18%. Heap and time overhead are mostly reasonable: typical heap overhead is 13% (Section 4.3) and typical performance overhead is 11% (Section 4.4). Due to its complexity and to a lesser extent its size (168K lines), perl took significantly more porting effort (three weeks), and slows down substantially (84%). However, HeapSafe helped us find six previously unknown bugs in perl.

4.1 Test Setup

Figure 4 summarizes the attributes of the 22 single-threaded programs we ported to HeapSafe. These programs include all the malloc/free based C benchmarks from SPEC2000 and SPEC2006 [30][9] (except libquantum because, as mentioned before, it uses complex numbers which are unsupported by CIL) and a few benchmarks from previous memory management studies [5, 16]. For benchmarks included in both SPEC2000 and SPEC2006, we used the SPEC2006 version. These benchmarks vary in size between 1270 lines (art) and 168,000 lines (perl). Some programs are very allocation intensive, as shown by the *allocs/s* column, including cfrac, grobner, espresso, parser, and perl. Others perform hardly any allocation, e.g. mesa, bzip2, mcf, and lbm.

The SPEC benchmarks were run on their reference input, for the other benchmarks we selected inputs with a reasonable runtime. Tests were performed on a 2.33GHz Intel® Core 2 Duo® with 2GB of memory.

We compare HeapSafe with the original C code linked with Doug Lea's malloc/free implementation v2.8.3 (on which HeapSafe's runtime is based) and with the Boehm-Weiser conservative garbage collector v6.7. Both of these libraries are compiled with multi-threading support disabled. The runtime libraries, the original code and the output from HeapSafe are compiled with gcc 4.1.2 at the -O3 optimization level.

[9] SPEC CPU is a single-threaded benchmark suite.

4.2 Porting the Benchmarks

All our benchmarks ran correctly with HeapSafe without any code changes being required. Code changes were only necessary in order to prevent HeapSafe logging error reports about benign frees.

Except for perl, these changes were made relatively easily. Time taken to port a program varied from a few minutes (equake, ammp, crafty), to three weeks (perl), with a typical program taking a few hours. No program required more than 2% of lines to be changed, and most changes were trivial (e.g. removing casts to `void*` or converting `free` to `ZFREE`).

The typical steps for porting a program to HeapSafe were:

- Modify the makefile so that it calls the debug version of HeapSafe rather than the standard C compiler;
- Build the project, and note any static warnings HeapSafe generates about the code;
- Add adjust functions and remove casts to `void*` and `char*` until you are happy with the set of warnings HeapSafe produces;
- Run the benchmark on test input, and search for the causes of the bad frees using HeapSafe's debug facilities that search the heap, stack and global variables for object references;
- Add scopes, ZFREE calls, type safety fixes and additional zeroing code until the program does not log any bad frees on its test input;
- In some cases (e.g. perl) it may be necessary to fix bugs in the original program source code in order to make the program pass; and
- In some other cases (e.g. espresso) it may be desirable to modify the code slightly in order to avoid needing to create an overly large delayed free scope; and
- Finally, improve performance by adding `nofree` annotations, removing unnecessary `memset` calls (because HeapSafe's malloc zeroes memory), and using typed allocation (Section 2.2).

Figure 4 summarizes the changes that we made to each of our benchmarks so they could run on their test input without logging any errors (perl still had some bad frees, as discussed below). The *changes* column shows the proportion of lines that were changed, the *S* column shows the number of lines used to start and end delayed free scopes, the *A* column shows the number of lines in hand-written adjust functions, the *F* column shows the number of `free` and `realloc` statements that were changed (typically removing casts or using ZFREE/ZREALLOC), and *O* represents all other changes (typically explicitly zeroing variables). The *F* changes were usually made very quickly using search and replace.

The fact that a benchmark runs on its reference input without logging any bad frees does not mean that it would not log bad frees on any input. It is likely that more changes would be necessary for these programs to avoid bad free warnings on all inputs. For the SPEC benchmarks, we did ensure that the official "test" and "train" inputs also ran without bad frees.

Most programs ported without any steps beyond those outlined above, but twolf, mesa, sphinx3, and, especially, perl required some other changes. In twolf, we fixed a space leak that was causing bad frees (the leaked objects referred to objects that were being freed). In mesa, we modified an explicit list destruction function to zero out pointers in the list rather than writing an appropriate adjust function for the list type. In bzip2 we fixed a logical error that caused a bad free (but which was not an actual bug). In sphinx3, we added a field to a data structure to discriminate between an union's fields. We describe the changes to perl in more detail below.

After eliminating bad frees, we reordered some frees in cfrac, parser, twolf, and hmmer to keep scopes smaller, reducing memory usage. Additionally, we replaced some calls to realloc in hmmer by a free/malloc sequence (the old data was not reused) to avoid a large space increase: Doug Lea's realloc implementation uses the mremap system call to reallocate large objects at a different virtual address but without a physical copy — HeapSafe's realloc cannot do this as it would be unsound.

Three programs, parser, sphinx3 and perl, used custom memory allocators for some allocations. In parser and sphinx3, we disabled this allocator (in both the original and HeapSafe versions). In perl, we left the custom allocators (which are type-specific allocators) in place, in part because perl relies on their existence to execute some destructors at program exit time. We used some low-level HeapSafe facilities (not described in this paper) to support perl's custom allocator . We believe that good support for custom memory allocators will be necessary in HeapSafe, and leave this for future work.

4.2.1 Porting Perl

The process of converting a program to HeapSafe can reveal memory bugs even when the test input does not access any freed objects: when HeapSafe reports a bad free, the programmer must investigate the dangling reference and either add a delayed scope around the free statement, or figure out when and where to zero out the offending reference. In the case of perl, we sometimes found instead that there was a code path where the dangling reference could still be used, which we confirmed by creating an appropriate test case. We found five such problems in perl. The sixth perl bug was a type safety error (accessing an array beyond its bounds) that prevented us from correctly zeroing out some dangling references. These bugs sometimes cause the wrong value to be returned, or allowed deallocated objects to be modified (because of perl's custom allocators, this often simply leads to another object of the same C type being modified, leading to incorrect behaviour but not crashes). One of the perl bugs

causes a number of bad free reports on the test input, but our fix for this bug (which we believe to be correct) causes a space explosion on another part of the test input.

Much of the porting effort for perl went in to understanding enough of its internals to figure out when and where to clear dangling references. This was particularly tricky for regular expressions, and appears to be a problem for perl authors too: two of our bugs were related to regular expression handling. Another major porting issue was understanding perl's various internal stacks and when to clear them — sometimes values beyond the stack pointer are still live... We ended up using explicit reference count operations for pushes and pops of pointers to perl's "save stack", for improved performance.

Beyond these changes, and bug fixes for the problems we discovered, we added a couple of benign leaks to simplify porting, added an "allocated type" field to perl's opcodes (perl sometimes changes an opcode's kind, but we may be able to remove our additional field with some extra work), and wrapped calls to perl's yacc-based parser in one relatively large delayed free scope (finding dead references on yacc's stack is difficult). We used the `norefcount` qualifier in three places: perl's method cache (see Section 2.1.2), perl's custom memory allocators, and perl's save stack (where we used explicit reference counting).

A final comment: most of the early bad frees reports in perl were not bugs. However, once we had understood enough of perl's internals to correctly place scopes and zero references correctly, many of the remaining bad frees corresponded to actual bugs.

4.3 Space

HeapSafe increases heap usage by a geometric average of 13% (Figure 4, Max Heap Usage columns). 12.5% of this is due to the reference count table,[10] and the rest is due to delayed free scopes keeping objects around for longer than otherwise. No program increased its heap usage by more than 23%. This largest increase (in twolf) was mostly due to the space needed to track scope contents, accounting for a 8.6% increase in space usage. The only other program where tracking scopes caused a significant space increase was cfrac (1.4%).

The heap usage of the Boehm-Weiser [7] garbage collector is less predictable. The geometric average heap increase for Boehm-Weiser (*GC* column) is 85%, but variations are large. Overhead is below 1% on mcf, but some benchmarks experience extremely large heap blowups, e.g., above 300% on h264ref; indeed milc and sphinx ran past the 3GB process size limit on Linux and were not able to complete their test runs. We attempted to use Boehm-Weiser's special allocation routines, `GC_malloc_atomic` for objects containing no pointers, and `GC_malloc_ignore_off_page` for objects which should only be reachable by a pointer to their first page, but these changes were not sufficient for milc and sphinx. These special allocation routines did prevent or reduce space explosions in ammp, hmmer and h264ref. For perl, we additionally had to disable interior pointer support (i.e., only pointers to the start of objects prevent garbage collection). It is not clear if perl would always run correctly under these circumstances.

The Boehm-Weiser garbage collector allows some tradeoff between space and time consumption by modifying the `GC_free_space_divisor` variable, which changes the collection rate. Setting this variable to 10 (its default value is 3) reduces heap overhead to an average of 48%, at the expense of increasing the runtime overhead to 14%. However, these figures cover a wide range of outcomes: grobner's space overhead drops to 77% and its runtime overhead increases to 175%, while parser's heap overhead and runtime are essentially unchanged.

4.4 Runtime

Figure 4 shows the runtime for each of our benchmarks. *Orig.* is the original CPU time (system and user) in seconds, as reported by the `times` system call. *HeapS.* is the increase in runtime due to HeapSafe, and *GC* is the increase in runtime experienced using the Boehm-Weiser conservative garbage collector [7]. All figures are the geometric average of three runs per input.

HeapSafe slows our benchmarks down by a geometric average of 11%. All benchmarks except perl (which has 84% overhead) slow down by 33% or less. In Section 4.4.1 below, we break down the sources of HeapSafe's overhead for four benchmarks with significant slowdown: grobner, espresso, parser and perl. We believe that with more engineering effort we can improve on these results significantly, since there are many promising optimizations that we have not implemented.

The Boehm-Weiser collector has significantly better performance, with results ranging from a speedup of 2% (lbm)[11] to a slowdown of 29% (grobner) — the mean slowdown (5%) is not fully comparable with HeapSafe's as some benchmarks did not complete. This is not completely surprising since the Boehm-Weiser collector is considerably more mature and garbage collection is normally considered to be faster than reference counting. HeapSafe's advantages lie not in its performance, but in its predictability, low space overhead, and the low risks involved in its use.

4.4.1 Sources of Performance Overhead

We break down HeapSafe's runtime overhead into the following categories:

[10] We reserve 512MB of address space for reference counts, but Linux allocates pages only on the first write, hence the 12.5% overhead. See Section 3.1.

[11] This speedup and ammp's slowdown are presumably not due to garbage collection as these programs perform very little memory allocation.

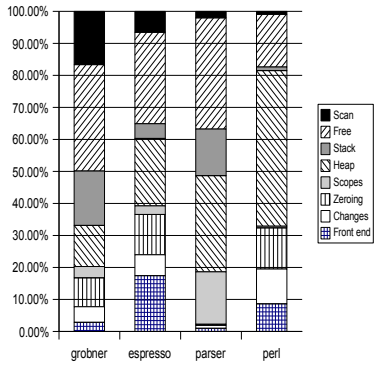

Figure 5. Breakdown of HeapSafe overhead.

- **Front End:** Compiling the program with the HeapSafe front end rather than the normal GCC front end.
- **Changes:** Effects of the various code changes (zeroing references, etc) needed to get rid of bad frees.
- **Zeroing:** Cost of zeroing newly allocated memory blocks.
- **Scopes:** Delaying object frees using scoped frees.
- **Heap:** Updating reference counts to record references from other heap objects.
- **Stack:** Maintaining the shadow stack for deferred reference counting. This overhead would be avoided with better compiler integration (Section 3.2).
- **Free:** Checking the reference count at `free` time, and removing references to objects pointed to by freed objects.
- **Scan:** Scanning the shadow stack for deferred references.

Figure 5 shows the proportion of the overhead that can be attributed to each of these sources for grobner, espresso, parser and perl, four of the highest overhead benchmarks. We obtained these numbers by running the benchmarks with some parts of HeapSafe and its runtime library disabled. Updating reference counts ("Heap") and freeing objects ("Free") are major costs in all four benchmarks. Unsurprisingly, grobner, espresso and parser, which have a high allocation rate (above 2M allocations/s), also have high overheads in the "Scan" and/or "Free" costs that are related to freeing objects.

The "Front End" and "Stack" costs are artifacts of our implementation of HeapSafe based on C-to-C translation. Based on Figure 5, we estimate that a native implementation of HeapSafe would have an overhead of 18% for grobner and espresso, 29% for parser and 77% for perl. These figures may be slightly optimistic, as a stack scan of the real stack may be more expensive than the simple linear scan of our shadow stack. We also believe that we could apply advanced static analysis to remove unnecessary reference count updates (the "Heap" cost), reducing HeapSafe's runtime overhead.

5. Related Work

Related work can be broadly divided into two categories: Garbage collectors automatically manage the freeing of objects, thus preventing bad frees but also causing changes to program behavior. Free-checkers check for bad calls to `free` in unmodified programs, but have less complete checking and much higher overheads than HeapSafe.

5.1 Automatic Memory Management

HeapSafe has different tradeoffs than garbage collection. In particular, it is easy to turn on and off (Section 2.4), is very predictable, and has lower space overhead. All operations in HeapSafe have a predictable overhead in space and time over a conventional malloc/free based system. Allocation takes time proportional to the size of the new object, deallocation of objects at the end of a delayed free scope takes time proportional to the sum of the sizes of freed objects and the number of live variables on the stack. There are no unpredictable pauses.

Since all memory is freed explicitly by the programmer, the programmer does not need to worry about unintended space leaks due to forgotten pointers. The increase in memory usage due to object retention in delayed free scopes is typically small, and is user-specified and predictable. HeapSafe's only other increase in heap size is a small constant factor due to reference counts. In contrast, garbage collection increases heap usage by retaining objects until the next collection, and typically requires a significant heap size increase to get good performance. Furthermore, as HeapSafe does not automatically free objects, it is tolerant of pointers held by foreign code.

Conversely, by removing the need to explicitly deallocate objects, garbage collectors simplify programming, avoid leaks due to forgotten frees, and prevent errors due to incorrect frees. In particular, the Boehm-Weiser collector [7] can be applied to existing C code with nearly no porting effort.

There is a vast literature about garbage collection [32, 17], including incremental garbage collectors that have short pause times and concurrent garbage collectors that collect garbage during program execution. Some garbage collectors allow the programmer to retain a degree of control over memory management. For example the Boehm-Weiser collector [7] allows one to free objects explicitly (and unsafely), and many collectors allow one to declare explicitly when garbage collection should take place, mostly avoiding pauses at inappropriate times.

Reference counting is usually seen as a mechanism for implementing garbage collection. When the reference count for an object reaches zero the object is freed automatically. This is in contrast to HeapSafe, where reference counting is

used only to check explicit memory management, and objects are never freed automatically. Recent reference counting implementations are very efficient [19]. It is very likely that HeapSafe could be made more efficient by borrowing techniques from these efforts.

When reference counting is used for garbage collection, it is necessary to address the problem that cycles of objects will have non-zero reference counts even if they are not reachable from the program roots [4]. HeapSafe does not need to worry about cycles since all objects are freed explicitly.

Bobrow [6] uses regions to aid in the freeing of cyclic structures within a conventional reference counting system. To allow a cyclic structure to be freed, one adds all elements of the structure to the same region, causing them to share one reference count, and allowing a reference-counting garbage collector to free the cycle.

OpenStep [24] (and thus also Cocoa [2]) allows a programmer to send an object an `autorelease` message, causing its reference count to decrement when the current autorelease pool is deleted. `autorelease` is used in a similar way to delayed free scopes — to allow short-lived dangling references to exist. The key difference is that `autorelease` requests a delayed decrement of the reference count, within a "zero means free" reference count system, while delayed scopes delay the actual `free` operation, within a `malloc`/`free` API.

5.2 Memory Management Checking

A number of tools exist that check the correctness of existing C code that uses `malloc` and `free`. Electric Fence [26], Purify [27], Valgrind [29], and DMalloc [31] all check for accesses to freed memory. Electric fence does this using page-protection hardware, Purify using code instrumentation, Valgrind using emulation, and DMalloc by checking memory for a special bit-pattern that is written to freed blocks. All three tools impose large performance and/or memory overheads[12], and all these techniques will fail to detect erroneous accesses to memory that has been recycled. In addition, DMalloc will only detect erroneous access after the fact, and after the program potentially went wrong. Many other tools [18, 3, 25, 20] use a combination of changed pointer representations, out-of-line metadata, and a separate process for checking C's memory safety. However, all have very high performance overhead. These tools are designed as bug finding tools for use at debug time, rather than as safety tools that can be enabled in deployed production code.

Several earlier proposals for checking dynamic allocation rely on the use of special pointer representations. Fischer and Leblanc [14] stores a unique *key* in objects and in pointers to object. The system checks that the pointer and object keys match on every dereference. In tombstones [21], pointers point to a handle (*tombstone*) that itself points to the object. The tombstone is marked as "dead" when the object is deallocated, allowing detection of dangling pointer accesses. Tombstones can be recycled using reference counting, or by using a key system similar to Fischer and Leblanc.

Dhurjati et al [11] ensure memory safety for C programs by segregating objects into separate "pools" based on their type (obtained through alias analysis). As a result, even dangling pointers are guaranteed to always point to an object of the correct type (albeit not the object the programmer expects). However, this system disallows most C casts, and will thus reject many programs.

Splint [13] allows programmers to annotate pointers with information such as `owned`, `only`, `dependent`, and `temp`, and checks that pointers are used consistently with these annotations. These annotations work well if the programmer is using their objects in an expected way and has taken time to annotate their program, but are less useful otherwise.

The RC system [15], earlier work by one of the authors, applies automatic reference counting to C programs that use region-based memory management. Like HeapSafe, RC checks that explicit deallocations are correct. An early version of HeapSafe was based on RC, and the annotations to find pointers (Section 2.2) are based on those of RC. RC's performance is better than HeapSafe's (overheads are below 11%), but RC only applies to programs using regions. Porting malloc/free programs to use regions requires substantial effort and is often impractical (Gay and Aiken did not modify any large malloc/free applications to use regions).

6. Conclusions

Although HeapSafe does not offer the performance or simplicity of a conservative garbage collector such as Boehm-Weiser, we claim it provides a practical way for maintainers of existing C code to ensure that their memory management is safe: it does not introduce unpredictable pauses or space-usage increases, it minimizes the risk of breaking existing code, and does not lock programs into a third-party tool. Its overheads for space and time are usually sufficiently low to leave checking on all of the time, and checking generally leaves execution behavior unchanged except for logging bad frees.

There are several issues that we intend to address in future work:

- **Performance:** We believe that, given sufficient engineering effort, we could significantly improve HeapSafe's performance.
- **Further memory checks:** It should be straightforward to extend HeapSafe to check for dangling pointers to stack frames, and for memory leaks.
- **API for new code:** Scoped frees seem to work well for existing code, but may not be the right API for new code. Also, many C programs use custom memory allocation. We plan to expose a low-level HeapSafe API that can be

[12] Even DMalloc, one of the most lightweight tools, imposes an overhead of over 400% on cfrac, even with all tests disabled.

used to check the safety of custom allocation with little effort.

- **Checked linear references:** Reference counts make it possible to verify dynamically whether one has exclusive access to an object. We intend to investigate if this can be used to make linear types, ownership types, and atomic sections easier to use or implement.
- **Concurrency support:** HeapSafe only has preliminary support for multi-threaded code.

HeapSafe is publically available at:

`http://memory.intel-research.net`

We encourage readers to download HeapSafe and apply it to their programs.

References

[1] ANDERSON, Z., BREWER, E., CONDIT, J., ENNALS, R., GAY, D., HARREN, M., NECULA, G. C., AND ZHOU, F. Beyond bug-finding: Sound program analysis for Linux. In *HOTOS XI* (2007).

[2] APPLE. Cocoa. http://developer.apple.com.

[3] AUSTIN, T. M., AND SOHI, S. E. B. G. S. Efficient detection of all pointer and array access errors. In *PLDI'94*.

[4] BACON, D. F., AND RAJAN, V. Concurrent cycle collection in reference counted systems. In *ECOOP'01*.

[5] BERGER, E. D. *Memory Management for High Performance Applications*. PhD thesis, University of Texas at Austin, 2002.

[6] BOBROW, D. G. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems 2*, 3 (1980).

[7] BOEHM, H., AND WEISER, M. Garbage collection in an uncooperative environment. *Software Practice and Experience* (1988), 807–820.

[8] CONDIT, J., HARREN, M., ANDERSON, Z., GAY, D., AND NECULA, G. Dependent types for low-level programming. In *ESOP'07*.

[9] DEUTSCH, L. P., AND BOBROW, D. G. An efficient, incremental, automatic garbage collector. *Communications of the ACM 19*, 9 (1976).

[10] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE'06*.

[11] DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. Memory safety without garbage collection for embedded applications. *Trans. on Embedded Computing Sys. 4*, 1 (2005), 73–111.

[12] ENGLER, D., AND ASHCRAFT, K. RacerX : Effective, static detection of race conditions and deadlocks. In *SOSP'03*.

[13] EVANS, D. Static detection of dynamic memory errors. In *PLDI'96*.

[14] FISCHER, C. N., AND LEBLANC, R. J. The implementation of run-time diagnostics in Pascal. *IEEE Transactions on Software Engineering SE-6*, 4 (July 1980), 313–319.

[15] GAY, D., AND AIKEN, A. Language support for regions. In *PLDI'01*.

[16] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference* (2002).

[17] JONES, R., AND LINS, R. *Garbage Collection*. Wiley, 1996.

[18] JONES, R. W. M., AND KELLY, P. H. J. Backwards compatible bounds checking for arrays and pointers in C. In *Automated and Algorithmic Debugging (AADEBUG'97)* (1997).

[19] LEVANONI, Y., AND PETRANK, E. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01*.

[20] LOGINOV, A., YONG, S., HORWITZ, S., AND REPS, T. Debugging via run-time type checking. In *FASE'01*.

[21] LOMET, D. B. Making pointers safe in system programming languages. *IEEE Transactions on Software Engineering SE-11*, 1 (Jan. 1985), 87–96.

[22] NECULA, G. C., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems 27*, 3 (May 2005).

[23] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CIL: Intermediate language and tools for the analysis of C programs. In *International Conference on Compiler Construction* (April 2002), Grenoble, France, pp. 213–228. http://cil.sourceforge.net/.

[24] NEXT. Openstep. http://www.gnustep.org.

[25] PATIL, H. G., AND FISCHER, C. N. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software Practice and Experience 27*, 12 (Dec. 1997), 87–110.

[26] PERENS, B. Electric fence. http://perens.com/FreeSoftware/.

[27] RATIONAL SOFTWARE. Purify: Fast detection of memory leaks and access errors. http://www.rational.com.

[28] SAVAGE, S., BURROWS, M., NELSON, G., AND SOBALVARRO, P. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Science 15*, 4 (1997).

[29] SEWARD, J., AND NETHERCOTE, N. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference* (2005).

[30] SPEC. SPECCPU 2000 and 2006. http://www.spec.org.

[31] WATSON, G. Dmalloc - debug malloc. http://dmalloc.com.

[32] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (1992).

[33] ZORN, B. The measured cost of conservative garbage collection. Tech. Rep. CU-CS-573-92, University of Colorado at Boulder, April 1992.