Purify:
Fast Detection of Memory Leaks and Access Errors

Reed Hastings and Bob Joyce
Pure Software Inc.

Abstract

This paper describes Purify™, a software testing and quality assurance tool that detects memory leaks and access errors. Purify inserts additional checking instructions directly into the object code produced by existing compilers. These instructions check every memory read and write performed by the program-under-test and detect several types of access errors, such as reading uninitialized memory or writing to freed memory. Purify inserts checking logic into all of the code in a program, including third-party and vendor object-code libraries, and verifies system call interfaces. In addition, Purify tracks memory usage and identifies individual memory leaks using a novel adaptation of garbage collection techniques. Purify produces standard executable files compatible with existing debuggers, and currently runs on Sun Microsystems' SPARC family of workstations. Purify's nearly-comprehensive memory access checking slows the target program down typically by less than a factor of three and has resulted in significantly more reliable software for several development groups.

1. Introduction

A single *memory access error*, such as reading from uninitialized memory or writing to freed memory, can cause a program to act unpredictably or even crash. Yet, it is nearly impossible to eliminate all such errors from a non-trivial program. For one thing, these errors may produce observable effects infrequently and intermittently. Even when programs are tested intensively for extended periods, errors can and do escape detection. The unique combination of circumstances required for an error to occur *and for its symptoms to become visible* may be virtually impossible to create in the development or test environment. As a result, programmers spend much time looking for these errors, but end-users may experience them first. [Miller90] empirically shows the continuing prevalence of access errors in many widely-used Unix programs.

Even when a memory access error triggers an observable symptom, the error can take days to track down and eliminate. This is due to the frequently delayed and coincidental connection between the cause, typically a memory corruption, and the symptom, typically a crash upon the eventual reading of invalid data.

Memory leaks, that is, memory allocated but no longer accessible to the program, slow program execution by increasing paging, and can cause programs to run out of memory. Memory leaks are more difficult to detect than illegal memory accesses. Memory leaks occur because a block of memory was *not* freed, and hence are errors of omission, rather than commission. In addition, memory leaks rarely produce directly observable errors, but instead cumulatively degrade overall performance.

Once found, memory leaks remain challenging to fix. If memory is freed prematurely, memory access errors can result. Since access errors can introduce intermittent problems, memory leak fixes may require lengthy testing. Often, complicated memory ownership protocols are required to administer dynamic memory. Incorrectly coded boundary cases can lurk in otherwise stable code for years.

Both memory leaks and access errors are easy to introduce into a program but hard to eliminate. Without facilities for detecting memory access errors, it is risky for programmers to attempt to reclaim leaked memory aggressively because that may introduce freed-memory access errors with unpredictable results. Conversely, without feedback on memory leaks, programmers may waste memory by minimizing `free` calls in order to avoid freed-memory access errors. A facility that reported on both a program's memory access errors and its memory leaks could greatly benefit developers by improving the robustness and performance of their programs.

This paper presents Purify, a tool that developers and testers are using to find memory leaks and access errors. If a program reads or writes freed memory, reads or writes beyond an array boundary, or reads from uninitialized memory, Purify detects the error *at the point of occurrence*. In addition, upon demand, Purify employs a garbage detector to find and identify existing memory leaks.

2. Memory Access Errors

Some memory access errors are detectable statically (e.g. assigning a pointer into a short); others are detectable only at run-time (e.g. writing past the end of a dynamic array); and others are detectable only by a programmer (e.g. storing a person's age in the memory intended to hold his height). Compilers and tools such as *lint* find statically-detectable errors. Purify finds run-time-detectable errors.

Errors detectable only at run-time are challenging to eliminate from a program. Consider the following example Purify session, running an application that is using the X11 Window System Release 4 (X11R4) Intrinsics Toolkit (Xt). The application is called my_prog, and has been prepared by Purify.

```
tutorial% my_prog -display exodus:0
Purify: Dynamic Error Checking Enabled. Version 1.3.2.
(C) 1990, 1991 Pure Software, Inc. Patents Pending.

...program runs, until the user closes a window while one of its dia-
logs is still up...

Purify: Array Bounds Violation:
Writing 88 bytes past the end of an array at 0x4a7c88 (in heap)
Error occurred while in:
 bcopy (bcopy.o; pc = 0x6d0c)
 _XtDoPhase2Destroy (Destroy.o; line 259)
 XtDispatchEvent (Event.o; pc = 0x33bfd8)
 XtAppMainLoop (Event.o; pc = 0x33c48c)
 XtMainLoop (Event.o; pc = 0x33c464)
 main (lci.o; line 445)

The array is 160 bytes long, and was allocated by malloc called from:
 XtMalloc (Alloc.o; pc = 0x32b71c)
 XtRealloc(Alloc.o; pc = 0x32b754)
 XtDestroyWidget (Destroy.o; line 292)
 close_window (input.o; line 642)
 maybe_close_window (util.o; line 2003)
 _XtCallCallbacks (Callback.o; line 294)
```

The Purify error message says that `bcopy`, called from `_XtDoPhase2Destroy`, is overwriting an array end, and that the target array was allocated by `XtDestroyWidget`, line 292.

```
        void XtDestroyWidget (widget)
         Widget widget;
        {
         ...
    292  app->destroy_list = (DestroyRec*)XtRealloc(
    293  (char*)app->destroy_list,
    294  (unsigned)sizeof(DestroyRec)*app->destroy_list_size);
        ...
        }
```

From this one can see that the target array is a destroy list, an internal data structure used as a queue of pending destroys by the two-phase Intrinsics destroy protocol. In order to understand why the end of the array is getting overwritten, one must study the caller of `bcopy`, `_XtDoPhase2Destroy`.

```
        void _XtDoPhase2Destroy(app, dispatch_level)
         XtAppContext app;
         int dispatch_level;
        {
         ...
    253  int i = 0;
    254  DestroyRec* dr = app->destroy_list;
    255  while (i < app->destroy_count) {
    256          if (dr->dispatch_level >= dispatch_level) {
    257                  Widget w = dr->widget;
    258                  if (--app->destroy_count)
    259                   bcopy((char*)(dr+1), (char*)dr,
    260                          app->destroy_count*sizeof(DestroyRec));
    261                  XtPhase2Destroy(w);
    262         } else {
    263                   i++;
                          dr++;
                }
         }
        }
```

Aided by the certain knowledge that a potentially fatal bug lurks here, one can see that the `bcopy` on line 259 is intended to delete an item in the destroy list by copying the succeeding items down over the deleted one. Unfortunately, this code only works if the DestroyRec being deleted is the first one on the list. The problem is that the `app->destroy_count` on line 260 should be `app->destroy_count - i`. As it is, whatever memory is beyond the destroy list will get copied over itself, shifted 8 bytes (the size of one DestroyRec) down. The resemblance to reasonable data would likely confuse the programmer debugging the eventual core dump.

Many people find it hard to believe that such an obvious and potentially fatal bug could have been previously undetected in code as mature and widely used as the X11R4 Xt Intrinsics. Certainly the code was extensively tested, but it took a particular set of circumstances (a recursive destroy) to exercise this bug, that might not have come up in the test suite. Even if the bug did come up in the test process, the memory corrupted may not have been important enough to cause an easily visible symptom.

Consider the testing scenario in more detail. Assume optimistically that the test team has the resources to ensure that every basic block is exercised by the test suite, and thus a recursive destroy is added to the test suite to exercise line

263 above. The memory overwriting will then occur in the testing, but it may or may not be detected. Unless the memory corrupted is vital, and causes a visible symptom such as a crash, the tester will incorrectly conclude that the code is performing as desired. In contrast, if the tester had used Purify during the testing, the error would have been *detected at the point of occurrence*, and the tester would not have had to depend on further events to trigger a visible symptom.

Thus Purify does not in any way remove the need for testing, but it does make the effort put into testing more effective, by minimizing the unpredictability of whether or not an exercised bug creates a visible symptom.

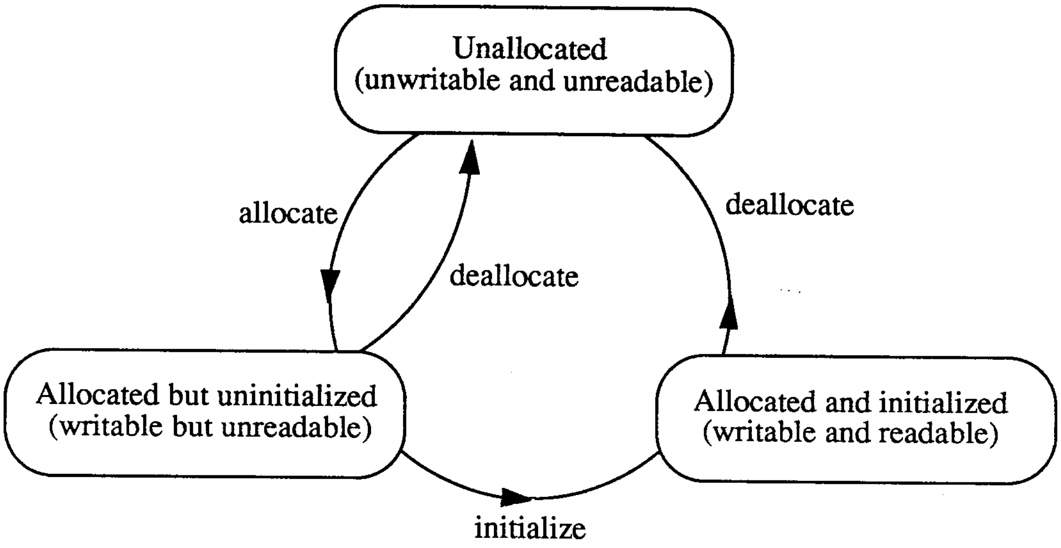
The effects of a library vendor missing a single memory corruption error like this Xt bug are quite serious: applications using the Intrinsics will occasionally trash part of their memory, and some percentage of the time this memory will be important enough to cause the application to later crash for seemingly mysterious reasons. Without a tool like Purify to watch over a library's use and possible misuse of dynamic memory, the application developer never knows if his application's crashes are his own code's fault or the fault of some infrequently exercised library code. This vulnerability and uncertainty is part of the reason that many developers still insist on "rolling their own" when it comes to utility routines.

3. Detecting Memory Access Errors

To achieve nearly-comprehensive detection of memory access errors, Purify "traps" every memory access a program makes, other than those for instruction fetch, and maintains and checks a state code for each byte of memory. Accesses inconsistent with the current state cause a diagnostic message to be printed, and the function CATCH_ME is called, on which the programmer can set a breakpoint.

Modifying the operating system to run a software trap upon every memory access would be prohibitively expensive, because of the context switch overhead. Instead, Purify inserts a function call instruction directly into a program's object code, before every load or store. The functions called, in conjunction with `malloc` and `free`, maintain a bit table that holds a two-bit state code for each byte in the heap, stack, data, and bss sections (the data and bss sections contain statically-allocated data). The three possible states and their transitions are shown in Figure 1.

FIGURE 1. Memory State Transition Diagram


Unallocated
(unwritable and unreadable)
allocate
deallocate
deallocate
Allocated but uninitialized
(writable but unreadable)
Allocated and initialized
(writable and readable)
initialize

A write to memory that contains any bytes that are currently in an unwritable state causes a diagnostic message to be printed; a similar message is printed if the program-under-test reads bytes marked unreadable. Writing uninitialized memory causes the memory's state to become initialized. When `malloc` allocates memory, the memory's state is changed from unallocated to allocated-but-uninitialized. Calling `free` causes the affected memory to enter the unallocated state.

To catch array bounds violations, Purify allocates a small "red-zone" at the beginning and end of each block returned by `malloc`. The bytes in the red-zone are recorded as unallocated (unwritable and unreadable). If a program accesses these bytes, Purify signals an array bounds error.[1]

To catch reads of uninitialized automatic variables, upon every function entry Purify sets the state of the stack frame bytes to the allocated-but-uninitialized state. In addition, each frame is separated with a red-zone to catch overwriting stack frame errors.

To catch array bounds violations in statically allocated arrays, Purify separates each static datum with a red-zone. Unfortunately some C code depends upon the contiguity of data statically defined together, and indexes directly from one static array into the middle of another. While this may seem a questionable practice, machine-generated code such as yacc parsers do make this assumption. Thus separating statically allocated arrays with red-zones has to be user suppressible, and Purify automatically suppresses it for yacc parsers.

To minimize the chance that accesses to freed memory will go undetected because the affected memory is quickly reallocated, Purify does not reallocate memory until it has "aged", and is thus less likely to still be incorrectly pointed into. The aging is user specifiable and measured in the number of calls to `free`.

In order to identify otherwise anonymous heap chunks, the call chain at the time `malloc` is called is recorded in the bytes that make up the chunk's red-zone. The depth of functions recorded is user specifiable.

Since there are three states, two bits are required to record the state of each byte. Thus there is a 25% memory overhead during development for state storage. In essence, Purify implements a byte-level tagged architecture in software, where the tags represent the memory state.

The advantage of maintaining byte-level state codes is that C and C++ programs can exhibit off-by-one byte-level errors[2] that would go undetected if a word-level state code approach was used. In fact, there is a continuum of choices here. Purify will catch the read of an uninitialized byte (representing a boolean flag in a struct, say), but will not necessarily catch an uninitialized bit field read. In the extreme case, Purify could maintain a two-bit state code for each *bit* of memory, giving a 200% overhead. In the authors' judgement, going from word tagging (6.25% overhead) to byte tagging (25% overhead) is quite worthwhile because of the additional error detection this change permits, but going to bit tagging (200% overhead) is not worthwhile.

An alternative scheme for state storage, that would completely forego byte and two-byte access checking, would be to store the state information directly in the data by using one "unusual" bit pattern to represent the unallocated state, and another to represent the allocated-but-uninitialized state. All other bit patterns would represent real data in the allocated and initialized state. This is the implementation strategy that Saber [Kaufer88], Catalytix [Feuer85] and various similar malloc_debug packages use. Byte and two-byte checking cannot be performed with this technique because there are no 8- or 16-bit patterns unusual enough to prevent false positives from occurring frequently.

---

[1] Since arrays in C & C++ are little more than a convenient syntax for pointer arithmetic, it is not possible to perform complete array bounds checking. In particular, errors of the form "`x = malloc(100); x[5000] = 1;`" will not always be caught because the address x + 5000 could point into another piece of valid memory. Purify allows the user to adjust the size of the red-zone to suit his particular space *vs.* thoroughness requirements.

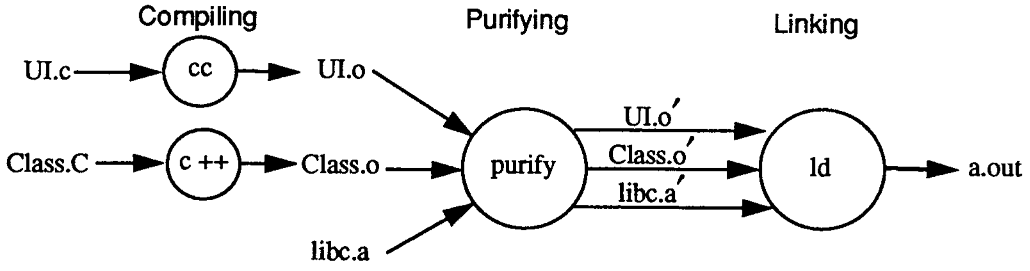
[2] Such as those caused by incorrect handling of a string's null terminating byte.

4. Object Code Insertion

Purify uses object code insertion to augment a program with checking logic.

Object code insertion can be performed either before linking or after linking. Pixie[3] is one program that does object code insertion after linking. Purify does it before linking, which is slightly easier, at least on Sun systems, since the code has not yet been relocated. Purify reads object files generated by existing compilers, adds error checking instructions without disturbing the symbol table or program logic, and feeds the output to existing linkers. Consequently, existing debuggers continue to work with Purified code.

FIGURE 2. Example Make



Another way to augment the program-under-test with the necessary checking logic would be to enhance the compiler to emit the required sequences, or to employ a portable pre-compiler. This would mean, however, that the programmer would have to recompile his files in order to use Purify, and that there would be no error checking in any libraries for which he did not have source code available.

Thus an advantage of object code insertion vs. a compiler or pre-compiler approach is setup performance. Since the re-translation from C or C++ to assembler is avoided, object code insertion can be much faster then recompilation. Our un-tuned implementation of object code insertion is more than 50 times faster (on a SPARC) than compilation.

Another advantage of object code insertion is convenience. The source for a large program lives in many directories, and the object code is already aggregated by the linker. To use object code insertion only the link target in the primary Makefile must change, instead of the ".c.o" compilation rules in every Makefile in the application.

Another advantage of object code insertion is multi-language support; many languages are quite similar at the object-code level. C and C++, for example, differ only in the encoding of the C++ names into "mangled names". Thus with the minor addition of a demangler to assist in the printing of symbol names, object code insertion programs such as Purify work with C++ as well as they work with C. We are currently exploring an ADA version.

A final advantage of object code insertion is completeness: *all* of the code, including third-party and vendor libraries, is checked. Even hand-optimized assembly code is checked. This completeness means bugs in application code (such as calling `strcpy` with too short a destination array) that manifest themselves in vendor or third-party libraries are detected. Also, serious bugs in third-party libraries (like writing into freed memory) can be detected, and the Purify messages can form the basis for highly-specific bug reports. Moreover, the detection or absence of such potentially fatal errors in a particular third-party library during the library's evaluation phase can increase the developer's knowledge of the quality of the code that will be included in his application.

---

[3] Pixie is a program that MIPS Computers Systems distributes to insert profiling code directly in an executable MIPS program.

The disadvantage of object code insertion is that it is largely instruction-set dependent, and somewhat operating system dependent—roughly like the back end of a compiler. This makes porting Purify to new architectures a substantial task.

5. Memory Leaks

Memory leaks are even harder than memory access errors to detect. The difficulty in detecting access errors is that the direct symptoms of such a bug may appear only sporadically—but a memory leak typically doesn't even have a direct symptom. The cumulative effects of memory leaks is that data locality is lost which increases the size of the working set and leads to more memory paging. In the worst case, the program can consume the entire virtual memory of the host system.

The indirect symptom of a memory leak is that a process' address space grows during an activity where one would have expected it to remain constant. Thus the typical test methodology for finding memory leaks is to repeat an action, such as opening and closing a document, many times and to conclude that there are no leaks if the address space growth levels out.

However, there are two problems with this methodology. The first problem is that it does not rule out that there simply was enough unallocated heap memory in the existing address space to accommodate the leaks. In other words the address space does not grow, but there does exist a leak. The assumption that testers have is that if the leak was significant enough to care about, it would have consumed all of the unallocated heap memory within the chosen number of repetitions and forced an expansion of the process's address space.

The second problem with this repetition methodology is that it is quite time consuming to build test suites that repetitively exercise every feature, and automatically watch for improper address space growth. In fact, it is generally so time consuming that it is not done at all.

Suppose, however, that a developer is sufficiently motivated to build a leak-detecting test suite, and finds that the address space grows unacceptably, due to one or more leaks. The developer still must spend a considerable amount of time to track down the problems. Typically, he would either (1) shrink the test suite bit by bit until the address space growth is no longer observed, or (2) modify `malloc` and `free` to record their arguments and perform an analysis of what was `malloc`'d but not-freed. The first technique is fairly brute-force, and can take many iterations to track down a single leak.

The second technique seems powerful but in practice has problems. In any given repetition loop, such as opening and closing a document, there may be `malloc` chunks that are `malloc`'d but legitimately not freed until the next iteration. Thus just because a chunk was `malloc`'d but not freed during an iteration does not mean the chunk represents a leak. It may represent a carry-over from a previous iteration. An improved technique [Barrach82] is to record the `malloc` and `free` calls for an entire program run, and look for chunks `malloc`'d but not freed. The problem with this is the existence of permanently-allocated data, such as a symbol table, that is designed to be reclaimed only when the process terminates. Such permanently-allocated data incorrectly show up as leaks, i.e. `malloc`'d but not freed, with this technique (2) and its variants.

Memory leaks are so hard to detect and track down that they are often simply tolerated. In short-lived programs such as compilers this is not serious, but in long-running programs it is a major problem. Consider how many hours have probably been spent eliminating leaks in the X11R4 server for Sun workstations. All that effort, yet dozens of leaks still exist—small, but leaks that accumulate into big effects. Here is one example session with the X11R4 server program, prepared by Purify, and running under the dbx debugger. It shows Purify catching the X server leaking one half of a megabyte from a single place, and the 10 minute sequence of events required to fix the leak.

```
tutorial% dbx Xsun
(dbx) run
Purify: Dynamic Error Checking Enabled. Version 1.3.2.
(C) 1990, 1991 Pure Software, Inc. Patents Pending.

...X server runs, we write more of this paper, then we interrupt the
server with control-C, and call the leak finding routine...

(dbx) call purify_newleaks()
Purify: searching for new memory leaks...

Found 43037 leaks.
There are 516752 leaked bytes, which is 35.9% of the 1437704 bytes in
the heap.

 12 (43026 times). Last memory leak at 0x35a058
516312 total bytes lost, allocated by malloc, called from:
 Xalloc (utils.o; line 515)
 miRegionCreate (miregion.o; line 279)
 miBSExposeCopy (mibstore.o; line 3458)
 miHandleExposures (miexpose.o; line 209)
 mfbCopyArea (mfbbitblt.o; line 283)
 miBSCopyArea (mibstore.o; line 1391)
 miSpriteCopyArea (misprite.o; line 999)
 ProcCopyArea (dispatch.o; line 1563)
 Dispatch (dispatch.o; line 256)
 main (main.o; line 248)
 start (crt0.o; pc = 0x2064)

 40 (11 times). Last memory leak at 0x36ee98
440 total bytes lost, allocated by malloc, called from:
 Xalloc (utils.o; line 515)
 miRectAlloc (miregion.o; line 361)
 miRegionOp (miregion.o; line 660)
 miIntersect (miregion.o; line 975)
 miBSExposeCopy (mibstore.o; line 3460)
 miHandleExposures (miexpose.o; line 209)
 mfbCopyArea (mfbbitblt.o; line 283)
 ProcCopyArea (dispatch.o; line 1563)
 Dispatch (dispatch.o; line 256)
 main (main.o; line 248)
 start (crt0.o; pc = 0x2064)
```

This example shows two leaks that have appeared so far in the current run of the X server. The first is the dominant leak, so let us walk through how to go from this information to finding the bug. The first leak has occurred 43026 times so far, and each time leaked 12 bytes. The first leak was probably not the responsibility of `Xalloc`, so we look at line 279 of miRegionCreate. It creates a region structure and simply returns it. So we turn to the caller of `miRegionCreate`: `miBSExposeCopy`, line 3458:

```
tempRgn = (* pGC->pScreen->RegionCreate)(NULL, 1);
```

A scan of the function confirms that `tempRgn` is never freed. A one line fix suffices.[4]

6. Detecting Memory Leaks

Memory leaks are allocated memory no longer in use. They should have been freed, but were not. In languages such as lisp and Smalltalk garbage collectors find and reclaim such memory so that it does not become a leak.

There are two parts to a garbage collector: a garbage detector and a garbage reclaimer. To achieve some of the benefits of garbage collection (lack of memory leaks) without the associated run-time costs or risks, Purify makes an important and novel change of focus. Instead of providing an automatic garbage collector, Purify provides a callable garbage *detector* that identifies memory leaks.[5] The garbage detector is a subroutine library that helps the programmer find and eliminate memory leaks during development. By using garbage detection to track down leaks, developers can benefit from garbage collection technology without suffering the normally associated delivery runtime costs.

Although the purpose is different, Purify uses an algorithm similar to the conventional mark and sweep. In the mark phase, Purify recursively follows potential pointers from the data and stack segments into the heap and marks all blocks referenced in the standard "conservative" and "pessimistic" [6] manner. In the sweep phase, Purify steps through the heap, and reports allocated blocks that no longer seem to be referenced by the program.

Identifying leaked blocks only by address would not help programmers track down the source of the leak; it would only confirm that leaks existed. Therefore, Purify modifies `malloc` to label each allocated block with the return addresses of the functions then on the call stack. These addresses, when translated into function names and line numbers via the symbol table, identify the code path that allocated the leaked memory, and often make it fairly easy for the programmer to eliminate the error.[7]

By moving the garbage collector technology from run-time to development, we are able to avoid the serious consequences of the fundamental problem with garbage collectors for C & C++, namely that there is always ambiguity in what is and what is not garbage. Our garbage detector separates the heap chunks into three classes:

1. chunks that are almost certainly garbage (no potential pointers into them), and
2. chunks that are potentially garbage (no potential pointers to the beginnings of the them), and
3. chunks that are probably not garbage (potential pointers do exist to the beginnings them).

---

[4] We don't mean to pick on X11R4 code; it's just widely-used, nearly-commercial-quality code. This leak, by the way, is also in X11R5.

[5] John Dawes, of Stanford University, co-invented this technology.

[6] See the following long footnote for an explanation of these terms.

[7] Obviously, better than fixing memory leaks would be avoiding them. Garbage collectors [Moon84] have been written for C and C++. Like other garbage collectors, they attempt to provide automatic and reliable storage management at some runtime cost. Generally they follow mark and sweep algorithms, and use the stack, machine registers, and data segment as root pointers into the heap. Since an integer in C is indistinguishable from a pointer, every plausible pointer, meaning every 32 bit word on most current machines, has to be considered a possible root pointer. It is assumed that the programmer is not "hiding" any pointers from the collector by such things as byte-swapping a pointer temporarily, or leaving the only reference to an object in a callback with an outside process. "Hiding" a pointer would cause the collector to reclaim something that was not yet garbage.

Since pointers cannot be distinguished from other types in C and C++, an integer with an unfortunate random value can "seem" to point to a chunk that otherwise might be garbage, causing that chunk to not be collected. This is why these collectors are often called "conservative". Such collectors are called "pessimistic" if they permit a pointer into the middle of a `malloc`'d chunk to anchor that chunk. The necessity of a collector being conservative and pessimistic leads to over-marking and under-collecting.

The fundamental flaw this introduces is that the larger a memory chunk becomes the *more important it is that it be collected* if it is garbage, because it's a significant resource, and the *less likely it is that it actually will be collected*, because it is more likely to be accidentally anchored by an integer value. This phenomenon is not limited to large single chunks; a doubly-linked list with many entries is vulnerable to the same error. Worse still, the error can be transient and unpredictable. Using a conservative garbage collector in the presence of large or interconnected chunks may work most of the time, and then grow without bound in a particular run, because of an unfortunate random value somewhere else in the program that "seems" to point into a chunk that is actually garbage. In broad terms, garbage collectors for C & C++ have excellent average case characteristics (high degree of de-allocation correctness), but fatal worst case characteristics (large chunks build up, recursively anchor enough memory to crash the program).

Each chunk is identified by its allocating call chain, and the developer uses his judgement on what and how to additionally free. If during the process of fixing the memory leaks the developer incorrectly frees a chunk prematurely, Purify's error detection will detect the eventual freed-memory access as soon as it occurs. Note that category three (3) above is all of the "live" allocated heap chunks, and can be used as profiling data to help understand where the heap space in a program is being used.

7. Previous Work

The difficulties of managing memory in C are well-known, and several attempts at addressing these issues have been made. Nevertheless, few C and C++ tools have succeeded in providing comprehensive solutions and none to our knowledge has addressed both memory leaks and memory access errors.

7.1 Malloc Debug

Malloc-debug packages are the most prevalent tool for finding memory access errors. These packages implement the `malloc` interface, but also provide several levels of additional error checking and memory marking. They can be useful for detecting a write past the end of a heap array, and require only a relink to use. Unfortunately malloc-debug packages do not detect errors at the point they occur; they only detect errors at the next `malloc_verify` call. Since `malloc_verify` has to scan the entire heap, it is expensive to call frequently. Further, these packages do not detect reading past the end of a heap array, accessing freed memory, or reading uninitialized memory.

Malloc debug packages do not provide any memory leak information.

7.2 Mprof

Mprof [Zorn88] provides information on a C program's dynamic memory usage to help programmers reduce memory leaks. Mprof does not provide any memory access checking.

Mprof is a two-phase tool requiring developers to exit the program under development before they can view the information Mprof provides. Developers can only obtain global statistics from Mprof; they cannot profile memory usage and leaks between arbitrary points of program execution, as they can with Purify. Mprof implements a "memory leak table" that identifies memory allocated but never freed. Unfortunately, this strategy confounds true memory leaks with memory allocated but not cleaned up during the exit process. Consider a symbol table that maps strings into symbols, in which the symbols are used as tokens and are never freed. When a program is about to exit, any time spent freeing memory is wasted, since the exit call will reclaim the process's entire memory. Thus, most Unix programs correctly call exit with large amounts of memory still in use. This memory does not constitute a leak, yet Mprof lists it as such. These false positives reduce Mprof's diagnostic value.

7.3 Saber-C and Saber-C++

Saber [Kaufer88] detects many run-time memory access errors in interpreted C and C++ source code. However, loading source code is time-consuming, and interpreting source code takes more than an order-of-magnitude longer than executing object code. Typically, programmers load only a few files in source form and load the rest in object form. As a result, many memory access errors remain undetected. Even if developers source load their entire application into Saber, it can not detect improper memory accesses from system libraries. For example, Saber does not detect the common case of calling `sprintf` with too short a destination string, even when called from interpreted code. Saber's interpreter also misses byte-level memory access errors, such as reading an uninitialized byte, due to the implementation of its state storage, discussed in section 3.

Saber does not provide memory leak information or memory usage statistics.

8. Measurements

The overhead that Purify introduces into a program is dependent on the density of memory accesses in that program. In the worst case, where the program does nothing but copy memory in a tight loop,[8] Purify's run-time overhead is a factor of 5.5 over the optimized C code. This compares with a factor of 3.2 slowdown for the same program compiled for debugging, and a factor of 300 slowdown for the same program running under a C interpreter.

Below we present data on Purify's overhead when used with two programs: the GNU compiler `gcc`, and the X11R4 demo program `maze` that animates the solving of a maze. The `maze` program was modified to remove its `sleep` calls. `gcc` is actually a small driver program, and `cc1` is the program that does the bulk of the work. It is `cc1` that was tested, although for simplicity we will refer to it below as `gcc`. The data was collected on a Sun SPARCstation SLC running SUNOS 4.1.1, and all times are real times.

	`gcc`	`maze`	average multiple
Run time[9] (seconds) optimized / Purified & optimized	26 / 81	117 / 178	2.3
a.out size[10] (kb)	815 / 1570	674 / 931	1.7
Max heap size[11] (kb)	1486 / 1775	540 / 608	1.2
Build time (seconds) link / Purify & link	7 / 35	5 / 24	4.9

The run-time overhead is mostly in the checking functions that execute before every memory access. The increased a.out size is due to the function call instructions inserted before every load and store. The heap size overhead is due to the red-zones kept around every heap chunk. The default red-zone policy, used in the test cases; gives each chunk a 16 byte initial red-zone and a 28 byte trailing red-zone. The build time overhead is half due to the Purifying process, and half due to the increased demands on the linker for resolving all of the references to the checking functions.

9. Summary

Purify provides nearly-comprehensive memory access checking and memory leak detection. It fits cleanly into the Unix file-processing paradigm and only requires adding a single word to the link-line of a makefile to use on an existing application. Importantly, Purify yields executables that are fast enough to use during the entire development and test process. For example, this paper was written using Frame while running under a Purified R4 X server, Purified window manager, and Purified xterms, all on Sun's bottom-of-the-line SPARCstation equipped with 12 Mb of memory. Purify's relatively low overhead, ease of setup, and thoroughness of error detection permits more robust software to be developed faster, yet it entails no overhead in code delivered to customers.

Purify can help bridge the gap between a program plagued by intermittent errors and that same program working robustly and continuously over long periods of time. Of course, Purify is not a panacea, and it does not result in bug-free code. Nevertheless, used in conjunction with good test suites Purify can result in significantly more correct and reliable programs, and increase the developer's knowledge and confidence in the code. Such progress creates programs that are less susceptible to catastrophic failure from small changes—making maintenance less risky, and testing less costly. Results from users of Purify working on large commercial programs have been very encouraging.

---

[8] Specifically, the program allocates one megabyte, initializes it to zero, and then performs 50 iterations of shifting the megabyte down one byte, by copying byte by byte.

[9] With gcc this is the time for `cc1` to compile and optimize the X11R4 client `xterm`'s file charproc.c. This file was picked at random to be the test case. With `maze` the times shown are the times to perform 20 iterations of solving the maze with the `sleep` calls between iterations removed.

[10] Measured with the `size` command.

[11] Measured with `sbrk(0) - &end`.

One of the great pleasures of C & C++ programming is being able to get the most out of the underlying hardware. Walking the tightrope of pointer arithmetic, for example, is very exciting but the downside is that most falls are fatal. Purify is the safety net that C and C++ always needed—it's there during development, but does not impair the ultimate performance.

10. Acknowledgments

Many people deserve thanks for their assistance on this project, but we wish to single out James Bennett for his outstanding guidance on how to present this work. Without him this paper would not have been.

11. References

[Barach82]	David R. Barach, David H. Taenzer, and Robert E. Wells. "A technique for finding storage allocation errors in C-language programs". *ACM SIGPLAN Notices*, 17(5):16-23, May 1982.
[Feuer85]	A.R Feuer, "Introduction to the Safe C Runtime Analyzer", *Catalytix Corporation Technical Report*, January 1985.
[Kaufer88]	Stephen Kaufer, Russell Lopez, and Sesha Pratap. "Saber-C An interpreter-based programming environment for the C Language". *Summer Usenix '88*, pp. 161-171.
[Miller90]	B. P. Miller, L Fredrickson, and B. So, "An Empirical Study of the Reliability of Unix Utilities", *CACM* vol 33, #12, December 1990, pp 32-44.
[Moon84]	David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984, pp. 235-246.
[Zorn88]	Benjamin Zorn and Paul Hilfinger. "A memory allocation profiler for C and Lisp programs." *Summer Usenix '88*, pp. 223-237.

12. Author Information

The authors can be contacted via electronic mail at hastings@pure.com and joyce@pure.com respectively. Their telephone number at Pure Software is (415) 747-0196. They are both graduates of Stanford's MSAI program.