To appear in *Software — Practice and Experience*

Low-cost, Concurrent Checking of Pointer and Array Accesses in C Programs*

Harish Patil and Charles Fischer

Computer Sciences Dept., University of Wisconsin —Madison

1210 W. Dayton St., Madison, WI 53706, USA.

e-mail: *{patil, fischer}@cs.wisc.edu*

Phone: 608-262-0018

Fax: 608-262-9777

June 6, 1996

*An earlier version of this paper was presented at the 2nd International Workshop on Automated and Algorithmic Debugging(AADEBUG'95) St. Malo, France, May 22-24, 1995.

Summary

Illegal pointer and array accesses are a major cause of failure for C programs. We present a technique called "guarding" to catch illegal array and pointer accesses. Our implementation of guarding for C programs works as a source-to-source translator. Auxiliary objects called guards are added to a user program to monitor pointer and array accesses at run time. Guards maintain attributes to catch out of bounds array accesses and accesses to deallocated memory. Our system has found a number of previously unreported errors in widely-used Unix utilities and SPEC92 benchmarks.

Many commonly used programs have bugs which may not always manifest themselves as a program crash but may instead produce a subtly wrong answer. These programs are not routinely checked for run-time errors because the increase in execution time due to run-time checking can be very high. We present two techniques to handle the high cost of run-time checking of pointer and array accesses in C programs: 1) "customization" and 2) "shadow processing". Customization works by decoupling run-time checking from original computation. A user program is customized for guarding by throwing away computation not relevant for guarding. We have explored using program slicing for customization. Customization can cut the overhead of guarding by up to half. Shadow processing uses idle processors in multiprocessor workstations to perform run-time checking in the background. A user program is instrumented to obtain a "main process" and a "shadow process." The main process performs computations from the original program, occasionally communicating a few key values to the shadow process. The shadow process follows the main process, checking pointer and array accesses. The overhead to the main process which the user sees is very low — almost always less than 10%.

KEY WORDS: memory access checking; program slicing; multiprocessor workstations

Introduction

Run-time checks can detect errors that cannot be detected at compile-time, including array bound violations, invalid pointer accesses, and use of uninitialized variables. Extensive run-time checking provided by diagnostic compilers incurs significant run-time costs. In [2], run-time checks were added to a C compiler. The code generated ran 10 times slower than the original code. Similar slowdowns are reported for commercially available run-time error checking systems such as *Purify* [3].

The high cost of run-time checks restricts their use to the program development phase when errors are frequent. As programs are fully developed and tested, they become more robust and their failure rate drops. Most runs of a well-tested program are error-free. Running such programs with run-time checking enabled slows the results of the computation, sometimes even by 1000%. To a typical user such delays are unacceptable, more so because there are no errors in most of the runs. Programmers therefore generally assume programs to be correct once the testing phase is over and disable run-time checks. This is dangerous because errors in heavily-used programs can be extremely destructive. They may not always manifest themselves as a program crash but may instead produce a subtly wrong answer. Even if an erroneous program crashes, it may be difficult to repeat the error inside a debugger. Further, debugging long running programs can be very time consuming. Undiscovered errors in heavily-used programs may not be rare; a study [4] has shown that

as many as a quarter of the most commonly used Unix utilities crash or hang when presented with unexpected inputs. Thus there is a strong case for running programs with checks routinely enabled, especially as computers assume ever-greater responsibility in commerce, communication, government, and transportation. Naturally, these checks should be as inexpensive as possible.

Pointer and array access errors top the list of run-time errors in a study [5] of over 100,000 Pascal program executions. The most common cause of program crashes in the previously mentioned study [4] of reliability of UNIX utilities written in C was also found to be pointer and array access errors. Apart from crashing the program or producing a wrong answer, unchecked pointer and array accesses provide a security hole as shown by the Internet worm [6].

Our project has two goals. First, to provide a run-time checking technique that provides extensive error coverage for pointer and array access errors in C programs. For this we have developed a source-to-source translation technique called '*guarding*.' We have focussed on the C language because the flexibility it allows in using pointers is unique; we believe the techniques in this paper can be easily adapted to other languages.We have a prototype implementation of guarding that can handle large, "real" C programs. Our prototype has been operational for almost two years. We have caught previously unreported errors in commonly used UNIX utilities and SPEC benchmarks. Most of these errors were missed by a popular memory access checking tool 'Purify'; mainly because Purify works on object files rather than source files.

Our second goal is to speed up the run-time checking so that it becomes affordable even for heavily used production programs. We have proposed a new way to address the high cost of run-time checking of pointer and array accesses. The basic idea is to decouple the run-time checking from the original computation so that run-time checking can be performed concurrently with the original computation or on its own when resources are available. We obtain a reduced version of the original program by deleting computations not contributing to pointer and array accesses. The resulting program can then be instrumented for run-time checking. We have explored the use of program slicing technology for reducing the original program.

We have also experimented with a technique called '*shadow processing*' to hide the cost of run-time checking by using idle processors in multiprocessor workstations. We partition a user program into two run-time processes. One is the main process executing as usual, without run-time checking. The other is a

shadow process following the main process and performing the pointer and array access checks. The shadow process performs all the computations needed to follow the main process. One key issue in the use of shadow process is the degree to which the main process is burdened by the need to synchronize and communicate with the shadow process. We believe the overhead to the main process must be very modest (say 5-10%) to justify the use of shadow processing for heavily-used production programs. We therefore limit the interaction between the two processes to communicating certain irreproducible values.

The rest of this report is organized as follows. Section "Guarding" presents details of our basic instrumentation technique for checking pointer and array accesses in C programs. Section "Decoupled computation and memory access checking" presents two ways to reduce the overhead of guarding by customizing the user program for pointer and array access checking. A technique to hide the overhead of guarding using idle processors in multiprocessor workstations is presented in Section "Concurrent memory access checking using shadow processing." In Section "Results" we discuss the performance of our guarding system and the effectiveness of various techniques we tried to handle the overhead of guarding. Later, the Section "Related work" is followed by the "Conclusions." A brief description of some of the errors discovered by our guarding system is presented in the "Appendix."

Guarding

We have developed a source-to-source translation technique for run-time checking of array bound violations and invalid pointer accesses in C programs. We call our checking technique *guarding*. It involves creating objects called guards to verify time and space bounds for pointers and arrays in the user program. These guards are used to check legality of pointer dereferences and array accesses. An illegal access may either violate a space bound (accessing an array element past the maximum index) or a time bound (accessing memory that has been de-allocated). Austin *et al* [7] have reported a similar technique. We call a user program instrumented for guarding a *guarded program*.

Implementation

Guarding was implemented on a Sun SPARCstation running SunOS Release 5.4 (Solaris 2.4). An overview of our prototype is shown in Figure 1.

Analyzing and tracking expressions involving pointers in C can be a formidable task. These expres-

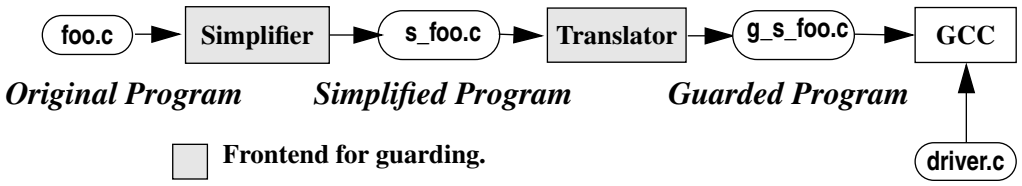

FIGURE 1. Overview of guarding system

sions may involve side-effects and multiple dereferences. Further, they can occur as loop conditions, array indices, actual parameters etc. A simplification phase was introduced to restrict the case analysis required for guarding. Our simplifier is a C-to-C translator whose output is a subset of C similar to the intermediate representation called SIMPLE from McGill university[8]. Simplification greatly reduces the number of cases to be analyzed by the translator phase – there are only 15 types of basic statements in any simplified program. However a large number of temporary variables may be introduced. These variables can increase the demands on register allocation. Hence a simplified program (compiled with `-O4` using `gcc 2.5.8`) typically runs around 1-2% slower than the original input program on a Sun SPARC 630 MP.

The translator in Figure 1 reads in a simplified program and produces a guarded program. The guarded program along with a driver routine can then be compiled by a native C compiler such as *gcc*.

Guards

Each pointer* `p` in the original program has a guard `G_p` in the guarded program. The guard for a pointer stores spatial and temporal attributes for the pointer's referent. The idea is similar to safe pointers used in [7]. However we store attributes separately from the actual pointer; this allows us to concurrently monitor pointers in one process using guards in another process. Operations on pointers in the original program lead to operations on guards in the guarded program. An array of pointers in the original program has an array of guards in the guarded program. Structures and unions containing pointers have objects containing guards in the guarded program.

Valid pointers in C contain addresses of data objects (including pointers) or functions. In programs that do not cast non-pointers into pointers, the origin of a valid object pointer can be traced back to either the address-of operator, `&`, or a call to a memory allocation routine such as **`malloc()`**. In either case, there is

*We will use the term pointer to include array references as well as ordinary pointers because when an array identifier appears in an expression, the array is converted from "array of T" to "pointer to T"[1].

an object the pointer is meant to reference. We call the object the *intended referent* of the pointer. The intended referent has a fixed size and a definite lifetime. It is clearly illegal to dereference an uninitialized pointer. Dereferencing an initialized pointer can be illegal for two reasons:

1. The memory location being referenced is outside the intended referent of the pointer. Dereferencing the pointer will lead to a *spatial* error.

   The attributes necessary to verify that a pointer dereference is spatially valid include the number of elements in the intended referent and the current position of the pointer inside the intended referent. The actual value of the pointer is unimportant.

2. The lifetime of the intended referent has expired (e.g. the pointer points to a heap or a local object that has been freed). Dereferencing the pointer will lead to a *temporal* error.

   We allocate a unique capability (called a `key`) for each memory allocation in the original program. These keys are stored in auxiliary data structures in the guarded program. Key values change as objects are allocated and freed. Two attributes are necessary to catch temporal access errors for a pointer `p` — a copy of the key of `p`'s intended referent and pointer to the location where the key is stored.

Given the declaration **`T`** `*ptr;` the guard `G_ptr` in the guarded program has fields storing spatial and temporal attributes for `ptr`. Fields of `G_ptr` are updated in the guarded program as the value of `ptr` changes in the original program. The fields of `G_ptr` are as follows:

`count:` The number of objects of type **`T`** being pointed to by `ptr`. If the intended referent of `ptr` is an array, this field will hold the number of elements of the array. If `ptr` gets cast into a pointer to another object type, this field will have to be recalculated.

`index:` A pointer in general may point to a collection of objects of a given type; pointer arithmetic is used to access a particular object in that collection. The field `index` in `G_ptr` denotes the offset of the current object being pointed to by `ptr`. For legal pointers, this is a non-negative value less than `G_ptr.count`. Pointer arithmetic on `ptr` leads to changes in `G_ptr.index`. *e.g.* in Figure 2, `G_p.index` is modified in the guarded program due to the statement "`p +=5`" in the original program.

`lock`: An identifying code used to check temporal legality of dereference of `ptr`.

Original code	Guarding code added
`int arr[10];`	`a_int_guard G_arr;/*initialize fields */`
`int *p;`	`a_int_guard G_p; /*clear fields*/`
`p = arr;`	`G_p = G_arr;`
`p += 5;`	`G_p.index += 5;`

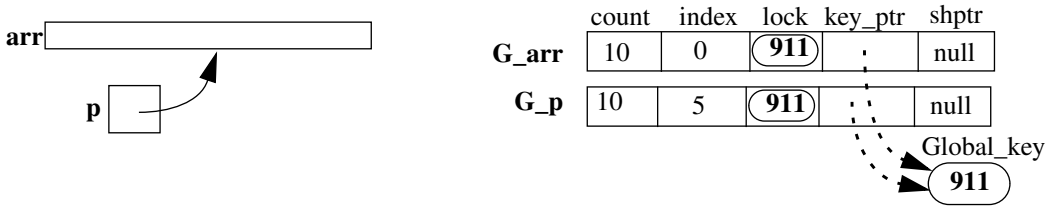

FIGURE 2. An example of guards for pointers and arrays

`key_Ptr`: This field points to the identifying key of an object.This code must match with the `lock` field of `G_ptr` for a dereference of `ptr` to be temporally valid. In Figure 2, `arr` is a global array, the field `G_arr.Key_Ptr` points to the location of the key for all global objects, `Global_Key`. Further, `G_arr.lock` has the same value (*911*) as that of `Global_Key`. After the assignment, `p = arr`, the intended referent of `p` is the same as that of `arr`, hence all the fields of `G_arr` are copied in `G_p`. Assignment of the fields `lock` and `Key_Ptr` is discussed in Subsections "Shadow stack" and "Shadow heap."

`shptr`: Pointers are data objects themselves, hence a pointer can reference another pointer. Each level of a multi-level pointer has a guard associated with it, and there must be a way to access each of those guards.The field `shptr` is used for that purpose. In Figure 3, the intended referent of `p` is another pointer `q`, `G_p.shptr` points to the guard of `q` *viz.* `G_q`. Thus the 2-level dereference `**p` leads to checking of two guards `G_p` and `*(G_p.shptr)` (which is `G_q`).

Original code	Guarding code added
`char **p,*q, c;`	`b_char_guard G_p; /* Level 2 guard */`
	`a_char_guard G_q; /* Level 1 guard */`
`q = &c;`	`/*set fields of G_q*/G_q.shptr = NULL;`
`p = &q;`	`/*set fields of G_p*/G_p.shptr = &G_q;`

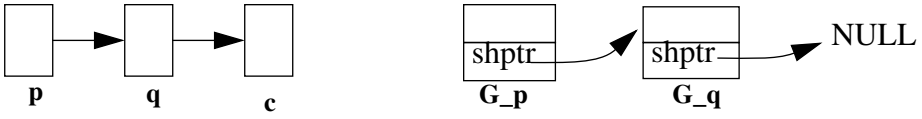

FIGURE 3. An example of guards for multi-level pointers.

In C, *invalid pointers* [1] can be created by casting arbitrary integer values to pointer types, by de-allocating the storage for the referent of the pointer, or by using pointer arithmetic to produce a pointer pointing outside its intended referent. It is legal to create or copy invalid pointers — attempts to dereference them are illegal. Thus pointer arithmetic and copying in the main program go unchecked.

Each pointer dereference implicit in `p[i]` or `*(p+i)` in the original program leads to two run-time checks in the guarded program:

```
check((unsigned)(G_p.index + i) < G_p.count)
check(G_p.lock == *(G_p.key_ptr))
```

The two checks are for the spatial and temporal legality of the dereference `p[i]` or `*(p+i)`. The first check is equivalent to "`0 <= (G_p.index+i) < G_p.count.`"

Shadow heap

An auxiliary data structure called the *shadow heap* is maintained in the guarded program to help check accesses to heap objects in the original program. The shadow heap is an expandable array of unsigned integers. Each heap object in the original program has a slot in the shadow heap containing an identifying integer that is a *key* for the object. After a dynamic allocation of an object *O* in the original program, a slot from the shadow heap is reserved for *O* until the time *O* is de-allocated. This slot stores the (essentially) unique key value for *O*. A list of empty slots arising due to de-allocations of objects is maintained along with the

shadow heap. These empty slots are reused later to avoid unbounded expansion of the shadow heap (much like a free-space list). When a pointer `p` points to a valid heap object *O*, `G_p.key_ptr` points to the shadow heap slot corresponding to *O*. Further, the key value in that slot matches `G_p.lock`. Key values are assigned using a global counter called *HeapKey* which wraps around to zero after reaching the maximum value (typically 2^{32}-1). Thus there is an extremely small chance (typically << 2^{-32}) that dereference `(*p)` of a pointer whose referent has been freed will go undetected. We shall consider the chance so small that it may be safely ignored. Actions on a dynamic allocation are explained below:

Original code	Guarding code added

`p = malloc(S)`	`G_p.count = S/aligned_sizeof(*p)`
	`G_p.index = 0`
	`G_p.key_ptr = Next_heap_slot()`
	`*(G_p.key_ptr)= G_p.lock = HeapKey++`

The function **`Next_heap_slot`**`()` returns the address of an empty slot in the shadow heap. If `p` is a multi-level pointer, **`malloc`**`(S)` results in allocation of a certain number (N) of pointers. Guards for these newly allocated pointers also need to be allocated using the statement "`G_p.shptr = calloc(N,sizeof(G_p))`" where `N` is `G_p.count`. If `p` is a single level pointer, a `NULL` value is assigned to `G_p.shptr`. Calls to **`calloc`** are handled very much like calls to **`malloc`**. A call **`realloc`**`(p,newsize)` leads to changes in `G_p.count`. A call **`free`**`(p)` leads to the checking of temporal and spatial legality of `*p`. "`G_p.index > 0`" indicates `p` currently points in the middle of an object; a warning message may be printed here. In addition, freeing of non-heap objects is detected by requiring that `G_p.key_ptr` points into the shadow heap.

Shadow stack

In C, it is illegal to dereference a pointer to a local variable of a function that has exited. To catch these dereferences, an auxiliary data structure called a *shadow stack* is maintained. It is a stack of unsigned integers. Each active frame in the run-time stack has a slot in the shadow stack containing its identifying key value. All local variables in a function share a slot and a key value. The key values are assigned using a global counter called *StackKey*. On a function entry, a slot gets pushed on the shadow stack, *StackKey* is incremented and its value gets stored in the newly pushed slot. On a function exit, the top slot from the shadow

stack is erased and popped. After the assignment `p = &var` in the original program, `G_p.key_ptr` in the guarded program is made to point to the shadow stack slot corresponding to `var`'s enclosing frame (global variables use a special `Global_Key` which is the same as the shadow stack slot for the function **`main()`**). As long as the frame containing `var` is active, `G_p.lock` will continue to match the key value in the slot pointed by `G_p.key_ptr`. After the frame containing `var` is exited, its shadow stack slot will be erased. If an attempt is made to dereference `p` now, the temporal check `(G_p.lock==*(G_p.Key_Ptr))` will fail. The shadow stack slot corresponding to an exited function will get reused on the next function entry (possibly to the same function) with a different key value (*StackKey* value at that time). Hence, dereferencing `p` will continue to lead to a temporal error.

Actions after an assignment `p = &var` are explained below:

Pointer operation	Guard operation

`p = &var`	`G_p.count = 1`
	`G_p.index = 0`
	`G_p.key_ptr =` **`<frame_slot`** `for var>`
	`G_p.lock = *(G_p.key_ptr)`

The value **`<frame_slot`** `for var>` is the address of the slot corresponding to `var`'s activation record in the shadow stack. If `p` is a multi-level pointer, `var` must be a pointer with its own guard `G_var`. In this case the statement **"**`G_p.shptr = &G_var`**"** is needed. Otherwise a `NULL` value is assigned to `G_p.shptr`.

`setjmp` and **`longjmp`** functions in C implement a primitive form of non-local jumps [1]. **`setjmp`**`(env)` records its caller's environment in the "jump buffer" `env`, an implementation-defined array. The function **`longjmp`** takes as its argument a jump buffer previously filled by a calling **`setjmp`** and restores the environment stored in that buffer. Many active frames on the stack may become inactive after a **`longjmp`**. We handle this by popping and erasing corresponding slots in the shadow stack before doing the corresponding **`longjmp`** in the guarded program.

Handling peculiar C language features

Our prototype requires that the input program does not cast non-pointer values into pointers. Casting a low level pointer to a higher level (*e.g.* casting an `int *` into `int **`) is also prohibited; as an exception casting `char *` returned by **`malloc()`** or **`calloc()`** to higher level pointers is allowed. We had to modify some of the test programs to handle casting of low level pointer to a higher level. Our translator currently detects pointer misuses and quits with an appropriate message. An alternative would be to flag the guard for a misused pointer to be unsafe at run-time and skip further checking for that pointer.

Multidimensional arrays in C can be reshaped (*e.g.* a two dimensional array of size `m`×`n` can be treated as a one dimensional array of size `m*n` or another two dimensional array of size `(m/2)`×`(2m)`). To uniformly handle such reshaping our simplifier converts multi dimensional arrays to one dimensional arrays. All the accesses to a multidimensional array are simplified into accesses to the corresponding uni-dimensional array.

There may be unions overlaying pointers with non-pointers in a C program. For example:

```
union {
   char *field1;
   int field2;
   struct s * field3;
}u1;
```

Suppose `u1` is a union described above. It is a "misuse" [1] to treat value of `field2` of `u1` as `field1`. We maintain a run-time tag indicating the type of the currently active field of a union in the guarding object for the union. The guard `G_u1` for `u1` in the example above looks as follows:

```
struct {
  char * tagname; /* currently active field */
  union {
   a_char G_field1;
   a_struct_s G_field3;
   }
}G_u1;
```

`G_u1.tagname` is set appropriately whenever a field of `u1` is assigned to in the original program. A reference to a field on `u1` in the original program leads to a verification of `G_u1.tagname` in the guarded program.

Functions with pointer arguments take corresponding guards as extra arguments in the guarded program. Functions returning pointers need to also return corresponding guards in the guarded program. A guard for a returned pointer is returned indirectly using an extra parameter.

Original code	Guarded code
`foo(char *p,int c);`	`guarded_foo(char *p,int c,a_char_guard G_p);`

`char * bar();`
 	
`q = bar();`
	`char * guarded_bar(a_char_guard *G_retval);`
	
	`q = guarded_bar(&G_q);`

Currently, we perform checks only for user defined functions for which source code is available and we provide a clean interface for external functions. There is no passing of guards for pointer parameters to external functions. Special handling is needed for external functions returning pointers. The external functions may either be "system calls" (as described in section 2 of the UNIX man pages) or "C-library routines" (as described in section 3 of the UNIX man pages). Calls to operating system kernel routines (system calls) are one source of non-determinism for programs. A system call is a well defined entry point into operating system code. The return value of a system call depends on the state of the operating system data structures at the time the call was made. If we want the guarded program to get the exact same return values from the system calls as the original program then we will have to instrument the original program to record the return values of system calls. From our experience with SPEC92 benchmarks, the number of values to be recorded are too few to cause any significant difference in the execution of the original program. However, we believe that letting the guarded program repeat the system calls is safe —as it will monitor *some feasible* run of the original program. Also it allows us to run a monitored program independent of the original program.

Currently, we support only single process programs. Programs calling **`fork`** or **`exec`** are not supported. We can envision creating a new guarding process for each child of the original process.

The library function **`qsort`** takes in a pointer to a comparison function provided by the user and calls that comparison function indirectly. Any calls to this user function from **`qsort`** will not be guarded since the call to **`qsort`** itself is not guarded. The current solution is to manually disable any guarding in every comparison function used in **`qsort`**. This disabling can be automated by hard-wiring it into our translator.

Decoupled computation and memory access checking

Checking pointer and array accesses is an expensive operation; it routinely slows down the input program typically 3-4 times. Most past attempts at reducing this overhead have concentrated on using compile time analysis to reduce the number of run-time checking assertions. There has been much work on eliminating redundant array bound checks in FORTRAN [9, 10]. Unfortunately techniques for FORTRAN can not be readily applied to C programs because of presence of pointers. Pointer arithmetic, equivalence of arrays and pointers, and aliasing in C programs require conservative assumptions to be made during compile-time analysis for eliminating run-time checks. We have implemented such a conservative analysis in our guarding prototype. We were able to remove up to 10% of the run-time checks from test programs.

We have experimented with an alternate technique for speeding up the program with run-time pointer and array checks. Instead of focusing on run-time checks to be *included* in a program we focus on the computation that can be *excluded* from the program. This technique is motivated by the fact that in practice a program is run with run-time checking enabled with the primary purpose of finding errors. The results of computations from a program with checking enabled are seldom of any interest. We propose to decouple pointer and array access checking from the original computation. The basic idea is to obtain a reduced version of the user program by deleting computations not contributing to pointer and array accesses. The reduced program is thus customized for pointer and array access checking. This reduced program can then be instrumented for run-time checking. We call the instrumented reduced program a *customized guarded program*.

We first implemented a simple scheme to generate a reduced program for guarding. We look at the calls in the user program to functions not defined by the user. These include calls to external functions — library calls and system calls. We term calls to functions such as **printf** which affect the external environment as "output calls." Some output calls such as writing into a file must not be repeated in the guarded program lest they interfere with the result of the user program. Most output calls clearly do not affect the pointer and array operations in the user program. We have modified the translator for basic guarding shown in Figure 1 for deleting output calls. We maintain a database of external calls. For each external function the database stores the category of the call — whether the call is an "output call" or not. Our modified translator

consults this database to delete the output calls from the user program and adds code for maintaining and checking guards to generate a customized guarded program. Deleting output calls in turn results in deletion of some more computation (via dead code elimination) contributing solely to those calls. We have found that this basic technique of deleting output calls can result in a customized guarded program that runs up to 23% faster than an embedded checking program (non-customized guarded program).

We found some test cases in which the customized guarded program obtained by deleting output calls takes almost as much time as the standard embedded checking approach. These programs do not spend much time in the output routines, hence the value of reducing output calls is minor.

The speed of guarding can be further improved by using slicing technology to remove computations not necessary for guarding. A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. We developed a prototype slicing tool using a slicing back end that is an offshoot of the Wisconsin Program-Integration System [11]. To customize the user program for guarding, we slice the user program using the pointer and array accesses as the criteria for slicing. The result is a smaller and faster executable program which performs only the computation affecting array indices and pointers being dereferenced. This program is then instrumented to generate a guarded program. A customized guarded program obtained using slicing can run more than twice as fast as an embedded checking program.

Our translator for generating a customized guarded program using slicing works in 4 phases. The first phase processes simplified files one-by-one. For each simplified file it produces two files: 1) a file containing the control flow graph (CFG) for the code in the simplified file along with node numbers for abstract syntax tree (AST) and 2) another file containing the slicing criteria in terms of AST node numbers for array accesses and pointer dereferences in the simplified file. The second phase of our translator reads in the pairs of files for all the simplified files produced by the first phase and calls a slicing routine in the slicing back end with some nodes in the CFG marked according to the slicing criteria. The slicing back end gives back results by marking the CFG nodes in the slice. For each simplified file, phase two of our translator then produces a file containing numbers of AST nodes in the resulting slice. Phase three processes simplified files one by one using the corresponding slicing results file from phase two and generates sliced simplified files.

Phase four is exactly like the original translator in Figure 1, except that it reads in a *sliced* simplified file. The result is a customized guarded program obtained using slicing.

Concurrent memory access checking using shadow processing

An interesting question to ask is 'When do we run the guarded program?' There are two alternatives: 1) Concurrently with the original program, using an extra processor. 2) On its own - before or after user program's execution.

The first alternative should work well for single user multiprocessor workstations that are becoming increasingly common. Vendors such as Sun and SGI offer multiprocessor workstations. Dual processor PCs have started appearing in the market. With rapid advances in microprocessor technology, high-performance microprocessors should soon routinely incorporate as many as four general-purpose central processing units on a single chip [12,13]. Our department has more than 60 dual processor workstations. We monitored 18 of those machines over a period of 24 hours using `condor'[14]. We found that 83% of the dual processor workstations that were active had a load average less than one. This implies that most of the time when a dual processor workstation was busy, one processor was running an application while the other processor was idle.

We have experimented with using idle processors in a multiprocessor workstation to perform concurrent guarding using a technique called *shadow processing*. Our translator for shadow processing is an extension of our translator for basic guarding shown in Figure 1. We create two processes for a user program. One is the *main* process, executing the user program as usual, without run-time checking. The other is a *shadow* process, following the main process and verifying the legality of the array and pointer accesses. The shadow process maintains guards for the pointers and arrays in the main process. The major advantage is that since memory access checking is taken out of the critical execution path of the main program; the overhead to the main process (which the user sees) is less than 10% in most cases. Thus there is no need to turn off run-time memory access checking; every execution of the main process can be monitored by a shadow process in the background.

A major plus in executing a guarded program on an extra processor is that the access error information is available concurrently with the results of the original program. However the main and the shadow pro-

cesses share the system resources such as memory. It is possible that for some memory-hungry programs the memory requirement of the shadow process may result in page thrashing, slowing down the main process. Also, if the shadow needs to exactly follow the execution path of the main process, certain irreproducible values such as interactive input and return values of system calls need to be read from the main process. This communication may further slow down the main process. We have modified our translator for shadow processing to consult a database to decide whether communication is needed on a particular external call. This database needs to be updated manually if new external calls are encountered. The translator adds code to the main and shadow process to communicate values using a circular buffer in shared memory (implemented using a system call **`mmap()`**). As stated in the section on guarding, we believe that it is safe to let the guarded program (running on its own or concurrently with the original program) repeat all the external function calls including the system calls.

Dedicating a processor for a shadow process may not always be feasible on heavily used multiprocessor workstations. For uniprocessor systems and heavily used multiprocessor systems the alternative of running the shadow program in isolation as resources become available is more practical.

Results

The test programs were selected from the SPEC92[15] benchmarks, uniprocessor versions of the SPLASH [16] benchmarks, and many commonly used utilities from SunOS 4.1.3. A utility called *fuzz* [4] was used to generate random input for the SunOS utilities. The SPEC and SPLASH benchmarks were tested with their reference inputs. For performance measurement we used *gcc* version 2.6.3 with optimization level -O4 to compile various versions of test programs. Execution times were measured on a dual processor Sun SPARCstation 20 running SunOS 5.4 (Solaris 2). This machine has two 66 Mhz Ross HyperSPARC with 256K L2 cache and 64MB of memory.

Errors uncovered

Run-time errors which do not crash programs can go unnoticed for a long time. Guarded programs report such errors as they occur. Programmers can use the feedback from the guarded programs to eliminate subtle bugs. We uncovered unreported errors in seven test programs which did not crash. These programs (with the number of errors in parentheses) were *decompress*(1) and *sc*(2) from SPEC92, *cholesky*(2) and *locus*(2)

from the SPLASH benchmarks, and *cb*(1), *ptx*(4), and *ul*(1) from SunOS. Four SunOS utilities *col*, *deroff*, *uniq*, and *units* crashed with random inputs. These were already reported to be buggy [4] in earlier versions of SunOS. However, we found new errors in these utilities as well. In all, we uncovered 19 errors in eleven programs. We include a brief description of some of those errors in the Appendix. We expect further testing will reveal errors in other widely-used programs.

Performance of Basic Guarding

We use programs from the SPEC92[15] benchmarks to report performance of guarding. Some characteristics of our test programs are described in Table 1.

Table 1: Characteristics of test programs

Program	# of files	# of lines	Description
alvinn	1	272	Trains a neural network using back propagation to keep a vehicle from driving off a road.
compress	1	1503	A data compression application that uses Lempel-Ziv coding to compress a 1MB file. (Modified for testing.)
ear	13	5237	Uses FFTs and other library routines to simulate the human ear.
eqntott	23	3454	Translates boolean equations into truth tables.
espresso	44	14838	An EDA tool that generates and optimizes PLA structures.
sc	8	8485	A spreadsheet benchmark.
xlisp	22	7741	A LISP interpreter.(Modified for testing.)

The core of our implementation is a source-to-source translator that generates a guarded program for an input C program. We compared the performance of our implementation of basic guarding with that of Purify which is a very popular commercial tool modifying object files for a variety of run-time checks. See Section "Related work" for more details on Purify.

Adding code for run-time checking increases static program sizes as shown in Table 2. In purified programs the increase in text segment size is proportional to the static number of loads and stores, in guarded

programs it is proportional to the static number of pointer operations. We do not know the implementation

Table 2: Increase in program sizes

Program	Original (# of bytes in thousands)		Purified (% increase)		Guarded (% increase)	
	text	data+bss	text	data+bss	text	data+bss
alvinn	144.6	463.6	181.2%	9.9%	8.1%	0.2%
compress	103.7	424.7	218.8%	10.8%	12.9%	0.2%
ear	181.7	24.1	156.8%	190.9%	52.6%	7.5%
eqntott	121.2	284.7	194.6%	16.1%	111.7%	467.0%
espresso	274.2	15.2	131.5%	302.6%	355.5%	32.9%
sc	253.4	97.4	133.9%	47.1%	145.3%	313.4%
xlisp	197.4	13.8	151.1%	333.3%	191.5%	65.2%

details of Purify but our guess is that the increase in data and bss segment sizes in purified programs is due to some internal data structures that a purified program maintains. For guarded program, the increase is proportional to the global pointer declarations in the original program since it leads to declarations of corresponding guards (of size 5 times the original pointer size) in the guarded program. The increase is very pronounced in case of *eqntott* and *sc*. Both these programs use parsers generated by a parser generator *yacc*. These parsers contain large global arrays of structures containing pointers leading to even larger arrays of objects containing guards in the guarded program.

Run-time checking incurs overhead in execution time as shown in Table 3. The increase in execution

Table 3: Increase in execution time (user + system)

Program	Original (time in seconds)	Purified (% increase)	Guarded (% increase)
alvinn (50 epochs)	25.6 s	774.6%	472.3%
compress	2.9 s	748.3%	113.8%
decompress	2.0 s	765.0%	145.0%

Table 3: Increase in execution time (user + system)

Program	Original (time in seconds)	Purified (% increase)	Guarded (% increase)
ear	307.7 s	589.1%	642.0%
eqntott	18.8 s	735.6%	1337.2%
espresso	7.9 s	1107.6%	648.1%
sc (loadc2)	38.6 s	613.5%	173.3%
xlisp	122.1 s	988.1%	775.6%

time for purified programs is proportional to the dynamic number of loads and stores in the original program. For guarded programs the increase is proportional to the dynamic number of pointer operations. The pointer operations include pointer arithmetic, pointer copying, passing pointer parameters in addition to pointer dereferences. The effect of having a large number of dynamic pointer operations is most apparent in the increased execution time for *eqntott*.

Run-time checking also increases the dynamic memory requirements of programs. In the absence of

Table 4: Increase in Peak Heap Usage

Program	Original (# of bytes in thousands)	Purified (% increase)	Guarded (% increase)
alvinn (50 epochs)	32.8	0%	124.7%
compress	16.4	0%	249.4%
ear	2514.9	1.3%	2.9%
eqntott	1826.8	0%	41.7%
espresso	221.2	85.2%	507.4%
sc (loadc2)	278.5	138.3%	153.0%
xlisp (9 queens)	655.4	0%	752.4%
xlisp (8 queens)	655.4	0%	122.5%

internal details of Purify our guess is that the increase in heap usage for purified program depends on the internal data structures that Purify* maintains at run time. The increase in heap usage for guarded programs is due to two factors: 1) dynamic allocations of structures/unions containing pointers and 2) sizes of shadow heap and shadow stack. A dynamic allocation of structures/unions containing pointers leads to a larger allocation of object containing guards. The effect of a large number of such allocations in apparent from the increased heap requirement for *espresso*. We maintain the auxiliary data structures 'shadow heap' and 'shadow stack' in heap at run-time in guarded programs. The size of the shadow heap is proportional to the total number of dynamic allocations in the original program. The size of the shadow stack is proportional to the maximum depth of the run-time stack of the original program. We see a large increase in the peak heap usage for *xlisp* (reference problem - 9 queens) because *xlisp* (9 queens) has more that 1 million activation records on the run-time stack at one point. For each of these activation records we have to maintain one slot (holding an integer key) in the shadow heap leading to increase in peak heap usage. When we ran *xlisp* on a shorter problem (8 queens) the peak heap usage came down drastically.

We found that the time to generate a guarded program is generally higher (up to 5-6 times in the worst case) than the time to generate a purified program. Increase in compilation time due to Purify's object level instrumentation can be high if an entire library has to be instrumented (as is the case with `libcurses` used in *sc*). Guarding increases processing time in two ways. First, the source-to-source translation with extensive analysis of pointer declarations and uses can be time consuming; our implementation works in two phases (simplification followed by translation) paying the cost of disk I/O for simplified files. Second, the guarded program takes much longer to compile and link than the original program because of the instrumentation added.

*It turns out that most of the dynamic memory a purified program uses is obtained using a call to **`mmap()`**. We do not report the amount of **`mmap`**ed dynamic memory because it is much harder to monitor than the amount of heap-allocated dynamic memory on Solaris 2.

Performance of Guarding with Customization

To judge the effectiveness of various overhead reduction technique using customization we compared the performance of customized guarded programs with those of basic, non-customized guarded programs.

Table 5: Effect of deleting output calls: user and system times

Program		compress	decompress	ear	queens (-a 11)
Original	user	2.3 s	1.3 s	302.5 s	1.0 s
	system	0.6 s	0.7 s	5.2 s	3.5 s
	total	**2.9 s**	**2.0 s**	**307.7 s**	**4.5 s**
Guarded	user	5.2 s	4.0 s	2277.9 s	3.7 s
	system	1.0 s	0.9 s	5.3 s	3.5 s
	total	**6.2 s**	**4.9 s**	**2283.2 s**	**7.2 s**
Customized Guarded (Without output calls)	user	4.8 s	3.4 s	2264.0 s	2.9 s
	system	0.4 s	0.2 s	0.5 s	0.1 s
	total	**5.2 s**	**3.6 s**	**2264.5 s**	**3.0 s**

Table 5 presents execution times (user and system) for a few of our test cases with and without output calls in the guarded program. Deleting output calls mainly reduces system time; there is also some change in the user time due to elimination of code rendered dead by deleting output calls. We noticed that deleting output calls did affect all the programs we tested; the effect is most pronounced for programs with a lot of output calls at run-time. An example of such a program is *queens* in Table 5 where deleting output calls made the customized guarded program run faster not only than the basic guarded program but also than the original program. This is test program from McGill university and with *-a 11* on the command line prints all the solutions to the 11 queens problem. The original program and the basic guarded program spend a lot of time in output calls most of which vanishes from the customized guarded program. We believe that *queens* is representative of certain class of programs such as graphics utilities that spend a lot of time in the output calls and can benefit the most from customized guarding by deletion of output calls.

We did notice that for some of our test cases deleting output calls from the guarded program has hardly

any effect on execution time. We are currently working on using slicing technology to customize programs as outlined in Section "Decoupled computation and memory access checking." We have been able to slice some of our test cases to obtain sliced customized guarded programs. Table 6 presents the execution time

Table 6: Effect of slicing on execution time (user + system)

Program	Original (time in seconds)	Purified (% increase)	Guarded (% increase)	Customized Guarded (no output) (% increase)	Customized Guarded (sliced) (% increase)
alvinn (50 epochs)	25.1 s	792.0%	483.7%	482.9%	282.1%
compress	2.9 s	748.3%	113.8%	79.3%	10.3%
decompress	2.0 s	765.0%	145.0%	80.0%	45.0%
queens (-a 11)	4.5 s	233.3%	60.0%	-33.3%	-60.0%
stanford (20 iterations)	10.4 s	659.6%	430.8%	426.0%	239.4%

overhead for the purified programs and 3 versions of guarded program. Using slicing to customize guarding looks very promising; it subsumes customization by deleting output calls. Slicing helped us to reduce execution time overhead of guarding for all the test programs we were able to slice.

Performance of Concurrent Guarding

Table 7 summarizes the execution time for some SPEC92 benchmarks for concurrent guarding using shadow processing. The overhead to the main process is mainly due to any memory conflicts with the shadow process. In case of *sc* communication of return values of library calls also contributes to the overhead; there was no communication for other test cases. The overhead indicates the delay in obtaining results of the original computation in the shadow processing environment. It is below 10% in most of the cases. To the average

user, the main process will appear almost indistinguishable from the original un-instrumented program.

Table 7: Concurrent Guarding using Shadow Processing: (user + system) time

Program	Original (time in seconds)	Embedded Checking		Shadow Guarding	
		Purified (% increase)	Guarded (% increase)	Main (% increase)	Shadow: without output calls (% increase)
alvinn (50 epochs)	25.6 s	774.6%	472.3%	**< 1%**	472.3%
compress	2.9 s	748.3%	113.8%	**10.3%**	89.7%
decompress	2.0 s	765.0%	145.0%	**10.0%**	85.0%
ear	305.4 s	589.1%	642.0%	**< 1%**	636.4%
eqntott	18.8 s	735.6%	1337.2%	**3.2%**	1328.2%
espresso	7.9 s	1107.6%	648.1%	**1.3%**	650.6%
sc (loadc2)	38.6 s	613.5%	173.3%	**6.7%**	95.1%
xlisp	122.1 s	988.1%	775.6%	**5.6%**	787.6%

Related work

The related work falls into three categories 1) a system called AE which motivated our work, 2) systems checking for memory access errors, and 3) concurrent dynamic analysis techniques.

Abstract Execution

Shadow processing was motivated, in part, by a tool called AE that supports *abstract execution* [17]. AE is used for efficient generation of detailed program traces. A source program, in C, is instrumented to record a small set of key events during execution. After execution these events serve as input to an abstract version of the original program that can recreate a full trace of the original program. The events recorded by the original program include control flow decisions. These are essentially the same data needed by a shadow process to follow a main process. AE is a *post-run* technique that shifts some of the costs involved in tracing certain incidents during a program's execution to the program that uses those incidents. In contrast, shadow processing is a *run-time* technique that removes expensive tracing from the critical execution path of a program and shifts it to another processor.

Other Systems

CodeCenter[18] is a programming environment that supports an interpreter-based development scheme for the C language. The evaluator in *CodeCenter* provides a wide range of run-time checks. It detects approximately 70 run-time violations involving illegal array and pointer accesses, improper function arguments, type mismatches etc. Interpretation of the intermediate code for supporting these checks is very expensive though; the evaluator executes C code approximately 200 times slower than the compiled object code.

Purify [3] is a commercially available system that modifies object files to, essentially implement a byte-level tagged architecture in software. It maintains a table at run-time to hold a two-bit state code for each byte in the memory. A byte can have a status of i) unallocated, ii) allocated but uninitialized, or iii) allocated & initialized. A call to a checking function is inserted before each load and store instruction in the input object files. This checking function verifies that the locations from which values are being loaded are readable (*i.e.* allocated and initialized) and the locations in which values are being stored are writable (*i.e.* allocated). Slowdowns by a factor of 5-6 are very common for Purified pointer intensive programs. Purify is very convenient to use because it works on object files and can handle third-party libraries for which source code may not be readily available. However the major disadvantage of working at the object level is that Purify can not track the intended referents of pointers. Any access to memory that is in an allocated state is allowed. This severely restricts the kinds of errors that Purify detects. For example, an out of bounds array access can go undetected if it accesses a location belonging to another variable. If a pointer's intended referent is freed and the memory is reallocated, dereferencing the pointer should lead to a temporal access error; however Purify is also unable to detect that error. However, Purify detects more types of errors than our system (e.g. detecting un-initialized memory read) and also performs memory leak detection.

Austin *et al* [7] have proposed translation of C programs to *SafeC* form to handle array and pointer access errors. Their technique provides "complete" error detection under certain conditions. They have reported execution time overhead in the range of 130% to 540% for 6 (optimized) test programs. Their experimental system requires the user to convert each pointer to a *safe pointer* using a set of macros. A safe pointer is a structure containing the value of the original pointer and a number of *object attributes*. An input C program, annotated with macros, results in a C++ program which combined with some run-time support performs pointer access checking. Guarding shares the "completeness" of error detection with *SafeC*. Unlike

the SafeC system, insertion of checks in our system is completely automated. Temporal access errors in SafeC are caught using a "capability" attribute which is an essentially unique value per object, much like the `lock` in shadow guards. However, checking temporal validity of a pointer access involves an expensive associative search in a capability database. Such a search is avoided in shadow guarding by adding the `key_ptr` field in guards. The value of the `key_ptr` in shadow guards also serves to determine the storage class of objects. Hence a separate "storage class" attribute, as in safe pointers, to catch freeing of global objects is not necessary.

We are aware of commercially available memory access checking tools called *BoundsChecker* from NuMega Technologies (*http://www.numega.com/*) and *Insure++* from ParaSoft (*http://www.parsoft.com/*). Presumably these tools also work at the source level like our system. Unfortunately we can not meaningfully compare our system with these tools as we have not come across any publications describing the internal details of these tools.

Concurrent dynamic analysis techniques

ANNA (Annotated ADA) is an Ada language extension that allows user defined executable assertions (checking code) about program behavior. An ANNA to ADA transformer that allows either sequential or concurrent execution of the checking code is described in [19]. Concurrent run-time monitoring is achieved by defining an ADA task containing a checking function for each annotation. Calls to the checking function are automatically inserted at places where inconsistency with respect to the annotation can arise. Like shadow processing, the ANNA to ADA transformer uses the idea of executing checking code concurrently with the underlying program. However, it generates numerous tasks per annotation, which may lead to excessive overhead. Executing user defined assertions seems like a good application for shadow processing.

Parasight [20] is a parallel programming environment for shared-memory multiprocessors. The system allows creation of observer programs ("parasites") that run concurrently with a target program and monitor its behavior. Facilities to define instrumentation points ("scan-points") or "hooks" into a running target program and dynamically link user defined routines at those points are provided. Threads of control that communicate with the parasites using shared-memory can be spawned. Parasight is an interactive system geared towards debugging of programs. The overhead incurred in the target program because of "hooking in" of

parasites is not an issue. Shadow processing can use some of the ideas from Parasight. In certain applications, the shadow process need not be active for the whole execution of the main program. It could be "hooked in" with an already executing main process when a processor becomes available, "spot checking" the main program. The shadow can start executing at certain well defined points, say at the entry of functions. The main process will have to leave a trail of indicators (in a shared buffer) indicating those points have been reached.

One approach to concurrent run-time checking is to use specialized hardware. Tagged hardware [21] can be used for type-checking at run-time. Watchdog processors [22] are used to provide control flow checking. The Makbilan architecture [23] has been proposed for non-intrusive monitoring of parallel programs in parallel. Unfortunately specialized architectures are not widely available and they may not be able to support the full range of desirable checks (e.g., pointer validity checking).

Conclusions

In this paper we presented an instrumentation technique called guarding for checking array and pointer access errors in C programs. We also presented two techniques to handle the high run-time costs of guarding using customization and concurrent monitoring.

Guarding works at the source level and hence can use the high level information about definitions and uses of pointers and arrays. This results in a better error coverage than tools working at the object level. Our implementation of guarding has found many previously unreported errors in many popular programs.

Customization is a technique that can help speed up any monitoring activity such as memory access checking or profiling. The basic idea is to obtain a reduced version of a user program customized to the monitoring activity. We explored two ways to customize a user program for guarding. First by deleting calls to library routines affecting the external environment (output calls) and second by using a prototype program slicing tool to throw away irrelevant computations from the user program. Deleting output calls benefits a class of programs performing a lot of graphics or terminal output the most. Slicing is a more general technique that can benefit a wider range of programs. Slicing in its most general form can be computationally expensive. However this high cost can be amortized over a large number of runs of the customized guarded program. By giving the proper slicing criteria monitoring only a selected part of the user program is also

possible.

Shadow processing is a technique that uses idle processors in multiprocessor machines to perform monitoring. We have used shadow processing for concurrent guarding. Current approaches to pointer access checking work sequentially, typically slowing a computation 3-4 times. Such high overheads make those approaches unsuitable for use with heavily-used programs. After programs are fully developed and tested, running them with embedded checks seems unacceptably slow. Most programmers turn off the checks; trading reliability for speed. Shadow guarding offers an excellent way out − a shadow process works silently in the background watching for run-time errors. Computations in the user program are performed by a main process. Error-free runs of the main process are only slightly slower than the original. Occasional erroneous runs lead to an error report (sometimes slightly delayed) from the shadow process. If the original program crashes, an error report from the shadow points to the root cause of the crash. Reports on errors that do not crash the original program can be extremely helpful in uncovering hidden bugs.

The guarding technique for C programs presented in this paper can be adapted to other languages. The overhead reduction techniques of customization and shadow processing can be applied to many monitoring activities other than checking for validity of memory accesses. We believe our techniques show great potential in improving the quality and reliability of application programs at a very modest cost.

Appendix: Bug Report from Guarding

We have used our system for pointer and array access checking to test programs from a variety of sources including SunOS 4.1.3 utilities, SPEC 92 benchmarks, and the SPLASH multiprocessor benchmarks. In the following we describe some of the errors we discovered. A utility called *fuzz* [4] was used to generate random input for the SunOs utilities. SPEC92 and SPLASH benchmarks were tested with their reference inputs, unless stated otherwise.

Benchmark: decompress

Source: SPEC92

Error #1: file compress.c

Function `getcode()` called from `decompress()`

```
1144    /* high order bits */
```

```
1145    code |= (*bp & rmask[bits]) << r_off;
```

Pointer `bp` is used to traverse global character array `buf[BITS]`. While decompressing the reference input the number of bits per code changes to 16. (This condition can be forced by changing `#define INIT_BITS to 16`). Sometime after this happens, `bp` dereferences one location beyond the array `buf` at line 1145.

Check: Put '`assert((bp-&buf[0])<BITS);`' before line 1145.

The purified program did not report any errors.

Benchmark: sc

Source: SPEC92

Error #1: File `sc.c`, in function `update()`:

```
213       /* Now pick up the counts again */
214       for (i = stcol, cols = 0, col = RESCOL;
215               (col + fwidth[i]) < COLS-1 && i < MAXCOLS; i++) {
```

An array bound violation occurs in the terminating condition of the for loop.The two operands of `&&` are in the wrong order. `i < MAXCOLS` must come before indexing `fwidth[i]`. When `i` becomes `MAXCOLS (40)` there is an array bound violation. (Occurs twice for input.ref/loada2)

Error #2: Found error in file `lex.c`, in function **`yylex()`**:

```
114       for (tblp = linelim ? experres : statres; tblp->key; tblp++)
115               if (((tblp->key[0]^tokenst[0])&0137)==0
116                && tblp->key[tokenl]==0) {
```

Array `tokenst` contains the current token and `tokenl` is the length of the current token. The for loop traverses a table of reserved words to see if the current token is a reserved word. Pointer `tblp` is used to point to various entries of a table of reserved words.The first part of the if condition checks if the first letter of the current reserved word and the current token are the same. (It uses some clever bit manipulation and properties of the ASCII character set to ignore case differences.) The second part makes sure that the current reserved word is not longer than the current token. For input token `goto`, `tokenl` is `4`. There is a keyword `GET` in the table `statres`. The first part of the if condition is satisfied (`(('G'^'g'&0137)==0)` is `TRUE`). For the second check, `tblp->key[tokenl]` accesses the 5th element of the current key `GET` which has length 4 (`'G','E','T','\0'`). Accessing the 5th element of an array of size 4 is clearly illegal.

The purified program misses these two errors. However, it reports 13 errors (uninitialized memory reads) in the libraries `libc` and `libcurses`.

Benchmark: cholesky

Source: SPLASH-1

Error #1: file util.U

Function `ReadSparse5()` defines `char type[3];`

```
166   type[3] = 0
```

Only valid indices are `0-2`. This error has been fixed in the latest release of the SPLASH benchmarks (SPLASH-2). The purified program did not report any error.

Error #2: file util.U

Function `ISort()`:

```
350 while (M.row[j-1] > tmp && j > lo){ ...
```

On some calls to **`ISort()`** when the value of `lo` is `0`, index of `M.row[]` becomes `-1` which is illegal. The statement should have been:

```
350 while (j > lo && M.row[j-1] > tmp){ ...
```

This error is still there in the latest release of the SPLASH benchmarks — SPLASH-2. The purified program did report this error and it also reported 9 uninitialized memory reads in the library function **`sscanf()`**.

Benchmark: locus

Source: SPLASH-1

Error #1: file ginput.U

Function `ReadSparse5()`:

```
201 if (NumberOfTerminals < 2){
...
203     free(NewWire);
204 }
```

`free(NewWire)` is followed by a dereference of `NewWire`:

```
222 NewWire->NumberOfGroups = IGrouptNumber;
```

The purified program did report this error.

Error #2: file timer.U

Function `ReportTimes()`

```
161 for(i=0; i <=Global->NumberOfWires; i++){
...
162 if(SegmentUsed[i] != 0) && (NetPinDistribution[i] != 0)){ ...
```

Valid indices for `SegmentUsed[]` and `NetPinDistribution[]` are `0-800`. For one of the reference inputs `Global->NumberOfWires` is `904` and invalid indices of the `SegmentUsed` and `NetPinDistribution` are accessed.

The purified program did report this error and it also reported one uninitialized memory read.

"locus" is not part of SPLASH-2.

Utility: col

Source: SunOs 4.1.3

Error #1: file col.c

```
229     c3 = *line;
```

Inside the program, there are many places where the pointer `line` is incremented and dereferenced without checking its validity.

The purified program does catch this error for some inputs. We have found a random input for which the purified program does not terminate (neither does the original program).

Utility: ul

Source: SunOs 4.1.3

Error #1: file ul.c

There are numerous array bound violations in function **`filter()`**. Array `obuf` is indexed using the variable `col`. If an input line contains >=512 characters `col` gets larger than the maximum index (511) allowed for `obuf`.

The purified program does catch this error for some inputs. We have constructed a sample input for which the purified program does not report this error.

Acknowledgments

We thank Bart Miller, Krishna Kunchithapadam, Tom Reps, Susan Horwitz, Genevieve Rosay, Miron Livny, Jim Pruyne, Jim Larus, and Steve Kurlander for helping with various aspects of our work. We also thank the anonymous referees for their suggestions to improve this paper. This work was supported by NSF grant

CCR-9122267. Some of the machines used during this work were supported by NSF grant CDA-9024618.

References

[1] S. P. Harbison and G. L. Steele Jr. *C - A Reference Manual*. Prentice Hall, 3rd edition, 1991.

[2] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software - Practice and Experience*, 22(4):825–834, April 1992.

[3] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, 1992.

[4] B. P. Miller, F. Lars, and B. So. An empirical study of the reliability of Unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[5] R. J. Leblanc and C. N. Fischer. A case study of run-time errors in Pascal programs. *Software - Practice and Experience*, 12:825–834, 1982.

[6] D. Seeley. A Tour of the Worm. In *Proceedings of the Winter USENIX Conference*, pages 287–304, January 1989.

[7] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[8] L. J. Hendren and B. Sridharan. The SIMPLE AST - McCAT compiler. ACAPS design note 36, School of Computer Science, McGill University, Montreal, Canada, 1992.

[9] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150, March-December 1993.

[10] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[11] T. Reps. Demonstration of a prototype tool for program integration. TR 819, Computer Sciences department, University of Wisconsin, Madison, Wisconsin, January 1989.

[12] P. Gelsinger, P. Gargini, G. Parker, and A. Yu. Microprocessors circa 2000. *IEEE Spectrum*, pages 43–47, October 1989.

[13] L. Gwennap. Multiprocessors head toward MP on a chip. *Microprocessor Report*, pages 18–21, May 9, 1994.

[14] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[15] B. Case. Updated SPEC benchmarks released. *Microprocessor Report*, pages 14–19, September 16, 1992.

[16] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[17] J. Larus. Abstract execution: A technique for efficiently tracing programs. *Software - Practice and Experience*, 20(12):1241–1258, December 1990.

[18] S. Kaufer, R. Lopez, and S. Pratap. Saber-C an interpreter-based programming environment for the C language. In *Proceedings of the Summer USENIX Conference*, pages 161–171, 1988.

[19] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *Computer*, 26(3):32–41, March 1993.

[20] Z. Aral and I. Gertner. High-level debugging in Parasight. In *ACM Workshop on Parallel and Distributed Debugging*, pages 151–162, May 1988.

[21] E. A. Feustal. On the advantages of tagged architectures. *IEEE Transactions on Computers*, C-22(7):1241–1258, July 1973.

[22] A. Mahmood and M. E. J. Concurrent error detection using watchdog processor - a survey. *IEEE Transactions on Computers*, C-37(2):160–174, February 1988.

[23] R. Rubin, L. Rudolph, and D. Zernik. Debugging parallel programs in parallel. In *ACM Workshop on Parallel and Distributed Debugging*, pages 216–225, May 1988.