CSE 598 Project Report

Memory leak detection using Electric fence and Valgrind

Team Members: Rahulkumar Thakkar (rhthakka@asu.edu) , Tarun Vyas (tvyas@asu.edu)

1. Introduction

1.1 What is Memory Leak?

A memory leak is a particular kind of unintentional memory consumption by a computer program where the program fails to release memory when no longer needed. When a program requests OS to allocate some memory by using calls such as 'malloc', 'calloc', new etc; and fails to give back the resource to the OS after using it, then we can say a memory leak has occurred. A memory leak may also happen if an object is stored in the memory and cannot be accessed by the running code in any condition.

1.2 Consequences of Memory Leak

Memory leaks are common errors in programming languages like C/C++ which do not provide automatic garbage collection techniques. Memory leaks are contributing factors to Software Aging which implies progressive performance degradation or sudden crash of software system due to exhaustion of OS resources. In user level processes running in modern OS, the memory consumed by the application will be released when the application terminates; which may not be serious in systems with large memories. Some of the scenarios that signify the serious cases of Memory leaks are when:

1. An operating system that does not release the memory consumed by the application when the application terminates. Eg : AmigaOS.[1]
2. The program is supposed to run for long times. Eg: Back-end Servers.
3. Embedded devices which may be left running for years. Eg: Network switches and hubs.
4. Portable devices which have very limited amount of memory.
5. Programs where new memory is allocated for continuing tasks such as rendering the frames on the screen in any video device.
6. The leak is present in the OS code or kernel itself.

Thus, detection of memory leaks is important and for quality assurance memory leaks reports may be submitted as unit test results.

2. Understanding tools to detect the Memory errors and leaks

‘Conservative’ garbage collection techniques can be added to the languages that do not provide these inherent capabilities and libraries for doing this are available in C and C++. In that case, the memory manager can recover the unreachable memory but cannot claim the memory that is still reachable by the code and thus potentially useful. This kind of memory management impose a performance overhead moreover, it does not detect all the memory errors that can cause leaks. Prevalence of such memory leak bugs has thus lead to the development of various tools that detect the memory leaks and different memory errors. These tools are used to improve the quality of the software ensuring its robustness. Some of the popular memory debugging tools are:

1. Valgrind
2. Memwatch
3. IBM Rational Purify
4. Insure ++
5. Electric fence
6. BoundsChecker etc..

In the forthcoming sections we focus on tools Electric fence, Valgrind and IBM Rational Purify and understand their ideological implementation.

2.1 Electric Fence

Electric fence (Efence) is memory (malloc) debugger developed by Bruce Perens. This tool is used to detect two common programming errors :

1. Accessing the memory region beyond the boundaries.

2. Accessing the memory region that is already freed.

Electric fence uses the virtual memory hardware to place an inaccessible memory page after (or before depending on config flags) the allocated memory block. Thus, when the code line will try to access the memory bound, then hardware will issue a segmentation fault by SIGSEGV signal and will terminate the process at the offending instruction, giving the exact location of error. This implies that the memory allocated will be at the end of the page table in case to detect the buffer overrun [5].

To use electric fence, user must first download the Efence library (libefence.a) and install it. After installation of the library, ‘-lefence’ can be used as an argument at the linking stage.

Eg : gcc –o output test.c –lefence

Note : -lefence cannot be used when malloc enhancement or malloc debugging libraries are used such as : -lmalloc / -lmalloc-debugger.

When using Electric fence for detecting memory overrun, Efence allocates at least two virtual memory pages per allocation. The address of the malloc will be the address of the inaccessible page minus the size of the memory block requested. This case is valid when the requested memory block size is a multiple of the word size. A scenario where the block size is not a multiple of the word size, compiler will do the necessary byte padding to maintain word alignment. So, Efence will actually not detect overrun if the accessed byte lies in the padded area.

```
Eg :    p = (char*)malloc(sizeof(char)*10);

        a[11] = 'a';                        // No error detected by Efence
```

To allow the memory allocation without the byte padding for detecting off -by one error, user can set EF_ALIGNMNET flag to 0. This is a global flag and can be used by defining as an extern variable in your source code and then setting it to 0. The following example well demonstrates the above scenario.

```
extern int EF_ALIGNMENT;

int main()

{     char *m; int i = 0;

      EF_ALIGNMENT = 0;     // Set this variable to detect over run with byte precision w/o compiler byte
                            //padding

      m = (char *)malloc(sizeof(char)*10);

      if (NULL == m) { printf("No memory\n"); exit(-1);

      }

      for (i=0; i<= 10; i++) {

            m[i] = 'a';     //over-run for 11th byte – segmentation fault issued

}

      return 0;

}
```

Similarly Efence provides more global flags called as configuration switches to detect different types of memory errors as per the user requirement. These config switches are:

1. EF_ALIGNMENT
   As seen earlier.

2. EF_PROTECT_BELOW
   Set this flag to 1 to enable E Fence detect the under-run errors. Thus, E Fence will place an inaccessible address page before the allocated memory block.

3. EF_PROTECT_FREE
   Set this flag to ensure that the block which has been freed will not be used further in allocation.

4. EF_ALLOW_MALLOC_0
   If this flag is set to any integer value, then E Fence won't trap the calls to malloc with size 0, which it usually does by default.

5. EF_FILL
   Set this flag to a value between 0 and 255, so that every memory block that is initialized will be set to this value.

2.1.1 Advantages

1. Electric fence uses the technique of inserting the inaccessible pages after or before the memory block allocated. This doesn't require any instrumentation for detecting the memory errors. Hence, it is fast compared to tools that using DBI.
2. Even a slight over-run or under-run for both reading and writing by a byte, results in segmentation fault.

2.1.2 Limitations

1. Efence is primarily just a malloc debugger that detects the under runs, over runs and use of freed blocks. Thus, other memory errors such as use of uninitialized memory location, memory leaks etc are not addressed.

2. The source code of Efence is not thread safe. Thus, it can't be used for multi threaded applications.

3. Efence uses at least two virtual pages per allocation. Thus, its memory overhead is very large." *Purify* does a much more thorough job than Electric Fence and does not have the huge memory overhead." – Bruce Perens (E Fence Author).

4. Efence doesn't prove to be a handy tool; in the sense, user must specify the flags/config switches to ask Efence to detect different kind of errors.

5. We found that E Fence also doesn't detect the off by one error in the structures which are not byte aligned in spite of using the EF_ALIGNMENT flag. Consider the example below:

```
extern int EF_ALIGNMENT;
struct abc{
    int a;
    char b;
};

struct enc {
    struct abc q;
    char p[10];
};

int main() {

    struct enc *var;
    EF_ALIGNMENT = 0;

    var  = (struct enc *)malloc(sizeof(struct enc));
    if (NULL == var) {
            printf("No memory\n");
            exit(-1);
    }

    var -> p[10] = 'b';          //accessing over bound
    printf("Done w/o errors\n");
    return 0; }
```

2.2 Memcheck tool from Valgrind

Valgrind originally was a tool for memory debugging, memory leak detection and program profiling. But, it has evolved as a generic framework for dynamic binary analysis. Valgrind, in essence is a process virtual machine that runs as an application in the host OS and supports a single client at a time.

Valgrind tool = Valgrind core + Valgrind tool plug-in. Ref [2]

Valgirnd works on a binary and dynamically recompiles it to translate to a simpler form called Intermediate Representation (IR). This is done by the Valgrind core. Now, a tool is free to work on this IR and add its functionality to perform a certain task. Memcheck, the most popular tool from the Valgrind suite works on this IR and adds the instrumentation code around the instructions to track the validity and addressability of the memory blocks that will be allocated at run time.

2.2.1 Meta Data

Memcheck maintains three kinds of meta data about the client under execution to check for memory errors and memory leaks. These are referred from [3].

1. A (Addressability) Bits
   Every memory byte is shadowed by single A bit. This bit is used to check if the process legitimately accesses the corresponding byte. 0 represents an unaddressable bye and 1 indicates addressable. These values will get updated as the memory blocks get allocated and freed at run time.
2. V (Validity) bits
   Every register or memory byte is shadowed with 8 V bits which mean that the byte is initialized /defined or not. Here 1 implies valid (defined) and 0 implies invalid (undefined) bit. Every write operation for the memory block will update these corresponding V bits.

3. Heap Blocks
   Memcheck records information of every heap block in a hash table. Along with the above meta data, it can detect the bad frees, repeated frees with memory leaks.

So, we observe that every memory byte has nine shadow bits (8 V bits and 1 A bit). Thus, there can be 512 possible states. But, the V bits will be checked for validity only when the A Bit says that the byte is addressable. This makes Memcheck robust enough to detect the errors at Bit precision level. Further we see how this shadow memory is actually implemented along with the data structures used.

2.2.2 Overview of implementation of Shadow memory

Memcheck's basic shadow memory data structure is primarily a two level table consisting of secondary map SM and a primary map PM. A pictorial view can be referred from the figure below.

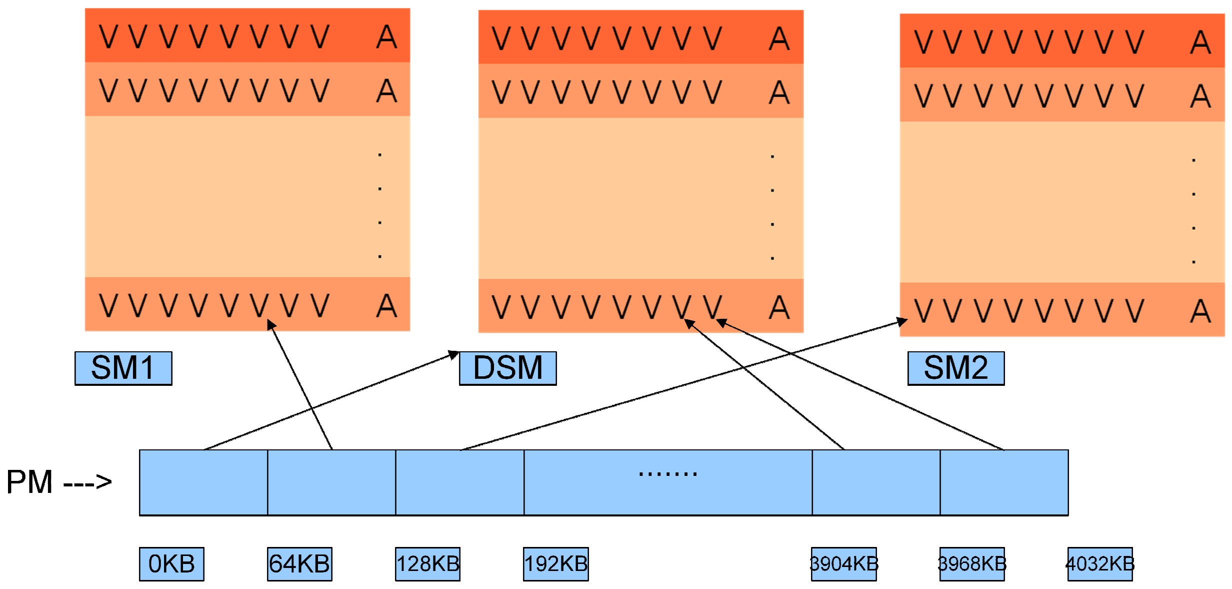


Fig 0: Two level table implantation of shadow memory. Ref [3]
Entries PM[1] and PM[2] which have been written point to corresponding SM whereas remaining entries point to NO-ACCESS DSM

The shadow memory data structures are defined below. These are valid only for 32 bit address space. The implementation for 64 bit is different. Also, these are not the exact implementations used in release versions of Memcheck but are used in the debugging mode of the Memcheck.

```
Typedef struct {

    U8 abits8[8192];      // 8K A Bytes/64K A bits              SM * PM[65536];    // Primary map

    U8 vbits8[65536];     // 64K V bytes                                          // Covering 4 GB

  } SM;
```

The address space is divided into 64K chunks each of 64KB. The primary map PM is an array of 64K entries and each entry is a pointer to the Secondary map SM. The secondary map contains the A Bits and V Bits. There is also a DSM (distinguished secondary map) which is marked entirely non-addressable. All the PM entries initially point to it. As shown in Fig 0, PM[1] and PM[2] point to some SM depending on the address and rest other entries point to DSM.

2.2.3 Single byte load and stores

We have seen the data structures for the shadow memory. Now, we have a look on how these shadow memory data structure is updated. Memcheck tool is built on the fundamental functions to load and store the shadow bytes. Below are the functions as described in Ref [3] for loading and storing single shadow memory bytes. The functions for multiple loads and stores are different.

1. Memcheck uses the high 16 bit of the address to locate the corresponding SM for given address. The function is :

```
SM* get_SM_for_reading(Addr a) {

            return PM[a >> 16];       // [31-16] bits of a to locate SM

    }

Here, Addr is 32 bit unsigned int representing address.
```

2. For storing Memcheck uses copy-on-write semantics. It checks if found SM points to DSM. If yes, then that SM is allocated and initialized.

```
SM* get_SM_for_writing(Addr a) {

        SM** sm_p  = &PM[a>>16];

        if (is_DSM(*sm_p))

                *sm_p = copy_for_writing(*sm_p);// allocate the SM and initialize
        return *sm_p;
}
```

3. Now, Memcheck uses the low 16 bits of the address to find the A bits and V bits within the SM. Loads are done with the following function: get_abit_and_vbits8. Extracting the V bits is straightforward but extracting A bits require some shifting and masking.

```
void get_abit_and_vbits8(Addr a, /*Out */ Uw * abit
                         /*Out */ Uw * vbits8 )
{
    SM * sm = get_SM_for_reading(a);

    // 13 bits of address used to determine 1 byte of 8 A bits
    U8 abits8 = sm->abits8[(a & 0xffff) >> 3];

    // last 3 bits used to determine the A bit in the byte
    *abit   = 0x01 & (abits8 >> (a & 0x7));

    *vbits8 = sm->vbits8[a & 0xffff];
}
```

4. Similarly, the function for the store case can be :

```
void set_abit_and_vbits8(Addr a, Uw abit, Uw vbits8)
{
        SM * sm = get_SM_for_writing(a);
        Uw shift = a &0x7;
        Uw i = (a &0xffff)  >> 3;
        Sm -> abits[i] = (sm->abits[i] & ~(1 << shift)) | ((abit &0x1) << shift);
        Sm -> vbyte[a & 0xffff] = vbits8 & 0xff;
}
```

Memcheck also records the location of every heap block with trace information in a hash table. In the end, the table is scanned for all the blocks to find any pointer references along with the shadow bit details, to detect any memory leaks and bad frees. Consider an instance if a pointer to a memory block is found in the table and the A bit of the address block is still 1 (addressable).This implies that a memory leak is detected.

2.2.4 Advantages

1. Memcheck is a robust tool to detect different kind of memory errors like over run, under run, uninitialized use, Memory leaks etc.

2. Memcheck works on bit-precion level whereas most other tools run on Byte precision level. Thus it can detect an error of using an uninitialized bit.

3. Memcheck uses Valgrind core which presents it with IR. Thus, a binary compiled on some platform can be run under Memcheck on other architecture.

2.2.5 Limitations

1. Memcheck is a dynamic analysis tool. The code will be checked for only the parts which it falls through. The rest of the code is not checked.

2. Memcheck has a big memory overhead and hence the code runs much slower. The mean slowdown factor using various benchmarks in SPEC2000 benchmark suite is between 22 to 23.5 as per the analysis done in Ref[2].

3. Memcheck catches the errors only below the stack.

```
 // This test provides an example of reading/writing inappropriate areas on the stack.  Note that valgrind
//only catches errors below

    int i;

     int * ptr = &i;

    ptr[8] = 7;                // Error, writing to a bad location on stack --not caught

    i = ptr[15];               // Error, reading from a bad stack location -- not caught

    i = ptr[-10];              // Error will be caught by Valgrind
```

4. It is incapable of detecting the static array bound errors.

3. Exploring some Memory leaks

Let's take a look at some common type of memory leaks.

- Common Leak error using global variable

```
int *pi;
void foo() {
        pi = (int*) malloc(8*sizeof(int));
        free(pi);
}
void main() {
        pi = (int*) malloc(4*sizeof(int));
        foo();
        pi[0] = 10;
}
```

In the above example, the user allocates 16 bytes of memory in main and saves the reference to a global variable pi. He then calls another function and overwrites the reference stored in pi by allocating another 32 bytes of memory. Then the 32 byte memory in heap is freed by using this reference. But, the 16 bytes of memory allocated earlier has no reference to it now and cannot be freed leading to a memory leak.

- Using memory area after releasing:

```
int i,*p,*temp;
p = malloc(10 *sizeof(int));
temp = p;
for (i=0; i< 5; i++) {
     temp = temp + i;        // temp points to intermediate location
}
free(p);
printf("%d\n", temp[i]);
```

In this code, the user allocates 40 bytes of memory first and then store its reference in two variables p, temp. Then inside a loop he moves the temp reference to the middle of the allocated block of memory. He then frees this memory using the another reference p. Next, he tries to dereference the freed memory which may lead to disastrous results.

Consider one more scenario:

```
int main() {

   char *p;
   int i;

   for (i = 0 ;i < 5; i++) {

    p = test(i+1);
    printf("Returned string is %s\n", p);
}
free(p); // will free only the last allocated value

 return 0;  }

char *test(int j) {

 int i;
char * temp;
temp = malloc(sizeof(char)*j);
for (i=0; i< j; i++) temp[i] = 'a';

return temp;
}
```

Here, the user calls a user defined function test in a loop. While in the loop he stores the reference of memory allocated by the test function. After the loop he frees the memory. But, he has only freed 5 bytes, whereas he calls the test function 5 times with different arguments leading to 10 bytes of leaked memory.

In the above examples we looked at some very simple examples of memory leak. It becomes very difficult to track these leaks as the code becomes huge and complex. We'll now look at a code which has a leak but isn't visible at the first glance.

```
int main (int argc, char *argv[])
{
        Char *lower;
        lower = to_lower (argv[1]);   // call the function
         while (*lower)
         putchar (*(lower++));
         return 0;
}

char *to_lower (const char *str)
{
        char *l = strdup (str);
        char *c;
        for (c = l; *c; c++)
        {
                if (isupper(*c))
                *c = to_lower(*c);
        }
        return l;
}
```

The above code the to_lower function calls strdup function which is defined in the standard string library.

```
char *strdup (const char *s) {
char *d = malloc (strlen (s) + 1);          // Space for length plus nul
if (d == NULL) return NULL;                 // No memory
strcpy (d,s);                               // Copy the characters
  return d;                                 // Return the new string
}
```

This function allocates memory equivalent to number of characters plus one bytes. The user doesn't know this and thus, doesn't free any memory. Thus we can conclude that a programmer should know the definition of functions which he is using; otherwise he may leak memory unknowingly. This code when run on Valgrind reports a leak of 6 bytes. Imagine if a similar program is run continuously in a real system leading to a large amount of leak and eventually program failure.

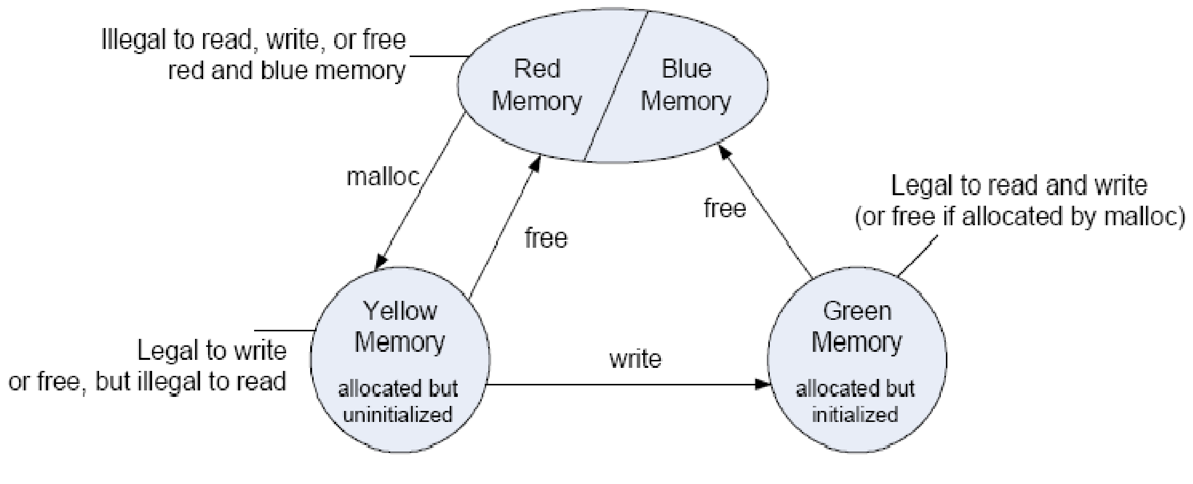
```
HEAP SUMMARY:

==3623==    in use at exit: 6 bytes in 1 blocks

==3623==   total heap usage: 1 allocs, 0 frees, 6 bytes allocated

==3623==

==3623==    6 bytes in 1 blocks are definitely lost in loss record 1 of 1

==3623==    at 0x4024F20: malloc (vg_replace_malloc.c:236)

==3623==    by 0x40AC07F: strdup (strdup.c:43)

==3623==    by 0x80484E4: to_lower (tough_example.c:12)

==3623==    by 0x8048557: main (tough_example.c:27)
```

4. Detecting Memory leaks with IBM Rational Purify

IBM rational purify is memory error and leak detecting tool for C and C++ programs. Purify classifies leaks into following types.

- MLK: Memory leak. When all pointers to a heap memory block are lost, the leak is reported as a MLK. This can happen when references are overwritten with other references or are lost when the program scope changes.
- PLK: Potential leak. This is reported when a pointer to the beginning of block is not there, but there is a pointer to the middle of a block.

Purify detects memory leaks by monitoring each byte of memory using two additional bits per byte of memory. It inserts its own instrumentation code into the program object code. Using the two bits, it defines 4 states into which a byte of memory can be in. The states are shown in the figure below.


Illegal to read, write, or free red and blue memory
Red Memory
Blue Memory
malloc
free
free
Legal to read and write (or free if allocated by malloc)
Yellow Memory
allocated but uninitialized
Legal to write or free, but illegal to read
write
Green Memory
allocated but initialized

The definition of states goes as:

- Red: Purify initializes the heap and stack as red. This memory is unaddressable and uninitialized.
- Yellow: Memory allocated using malloc and friends or new is yellow. This memory has been allocated, so the program owns it, but it is uninitialized. This memory should not be read.
- Green: When something is written into yellow memory it is moved to this state. This means that the memory is addressable as well as initialized.
- Blue: A freed memory block is moved to blue state. This means that the memory is initialized, but is no longer addressable

Limitations of Purify:

Since purify works at byte level; it may report false negatives when compared to Valgrind which works at bit level. The following example runs fine with Purify but not with Valgrind.

```
void  set_bit (int *arr, int n){
                arr[n / 32]  |=  (1 << (n % 32));
        }
int get_bit (int * arr, int n){
                return 1 & (arr[n/32] >> (n % 32));
}
void main(){
                int  *arr = malloc(10*sizeof(int));
                set_bit(arr,10);
                printf("%d\n",get_bit(arr,10));
}
```

In this program, the user has allocated 40 bytes of memory and then initializes 10 bit (in the second byte) of this allocated memory. He then tries to read this 10th bit but gets an error when run on purify. This happens because purify detects this as a yellow memory (addressable but un-initialized) which is not in reality, since a bit is initialized and then read. Valgrind doesn't suffer from this shortcoming. But the expense borne by Valgrind is high memory over head.

5. Kernel Memory leaks

The Linux kernel is a huge data structure and memory in kernel is freed only when the system is rebooted. But the tools which we have discussed so far only work on the user space programs. For Kernel, so far we have only two methods to detect memory leaks.

- kmemleak

- a patch developed by Catalin marinas

To enable leak detection in the kernel, CONFIG_DEBUG_KMEMLEAK has to be enabled in the /usr/src/linux-headers-<kernel_version>/.config file. After the enabling this option, the kernel has to be rebuilt. This results into a kernel thread scanning the memory periodically and reporting leaks in the /sys/kernel/debug/kmemleak file.

One can also disable the leak scan by setting the off option in the /sys/kernel/debug/kmemleak file. The summary of algorithm used is:

- Initially the memory region is all marked as white meaning that all objects are leaking.
- The references obtained by kmalloc and friends are stored in a priority search tree.
- While scanning the memory, an object to which a reference is found is marked as gray indicating that it is no more a leaked object.
- In the end, the objects which are still marked white are considered to be leaking and are reported.

Since, kmemleak is scans the memory, there may be performance issues resulting into slowdown. On systems with Symmetric Multiprocessors, a reference may be stored in register of any of the processors. To deal with false positives, the kernel thread must scan registers of all the processors so that a reference is not deemed to be leaking though it was present in the register of some other processor.

6. Short insight into Static Analysis using Secure programming Lint

As we have seen that the dynamic analysis is done to detect the run time errors in the code. Static analysis on the other hand works directly on the source code before compilation. Static analyzer helps in standardizing the process of finding the common programming mistakes which help in reducing the potential run time bugs. Static analyzer will help in catching the bugs in the complete source code as opposed to run time debuggers which catch errors in the parts which get executed.

Uses of Splint:

Splint is a tool for statically checking C programs for security vulnerabilities and common programming mistakes. The common errors detected by Splint may be noted as below:

1. Dereferencing a possibly NULL pointer.
2. Using possibly undefined storage or returning storage that is possibly not properly defined.
3. Type mismatches with greater precision than most of the C compilers.

4. Violations of information hiding. Memory management errors like dangling references.
5. Buffer over-flow vulnerabilities.
6. Problematic control flow, likely infinite loops, fall through switch cases etc.

As per Ref[5], Programmers are required to put efforts in annotating programs to get better checking results. A representational effort-benefit curve for using Splint is shown in Figure 1 below. Splint is designed to be flexible and allow programmers to select appropriate points on the effort-benefit curve for particular projects. As different checks are turned on and more information is given in code annotations the number of bugs that can be detected increases dramatically.

Splint Manual

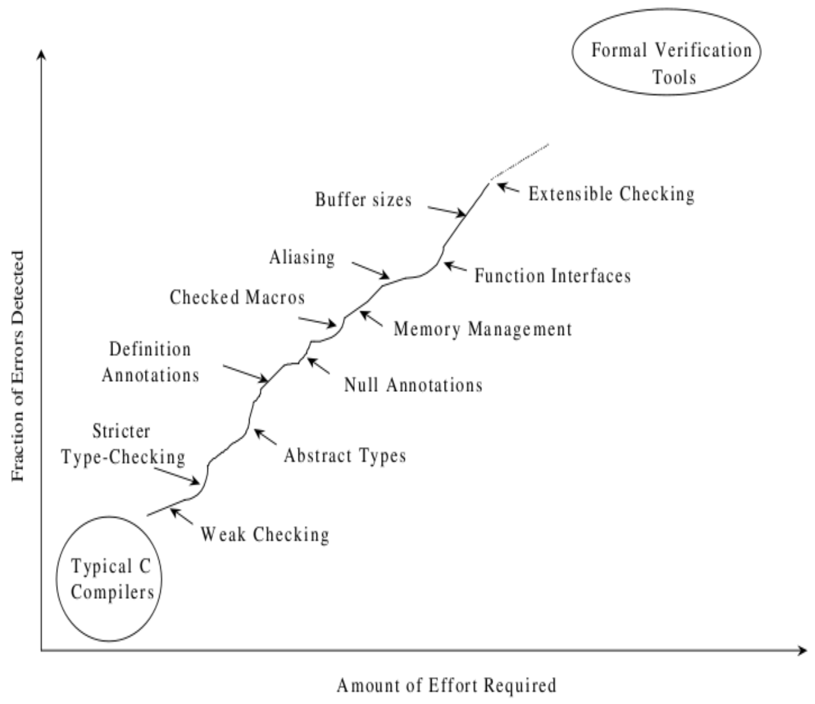


Figure 1. Typical Effort-Benefit Curve

A simple usage of splint with the code can be shown as:

```
int * foo(int j) {

    int i = 5;
    return &i;
}
int main() {

    int *result = NULL;

    result = foo(2);

    printf("Result is %d\n", * result);
    return 0;
}
```

Command to use splint to avoid obvious warnings is :

```
$> Splint -weak test.c
```

```
test.c: (in function foo)

test.c:10:9: Stack-allocated storage &i reachable from return value: &i

  A stack reference is pointed to by an external reference when the function
  returns. The stack-allocated storage is destroyed after the call, leaving a
  dangling reference. (Use -stackref to inhibit warning)
Finished checking --- 1 code warning
```

7. Overhead in Valgrind

Since Valgrind employs Dynamic Binary Instrumentation to detect memory leaks, it may incur a lot of overhead. We have run the following 5 benchmarks from the SPEC2006 benchmark suite to test the overhead that Valgrind incurred in each of them. The machine specifications on which these tests were conducted are: Intel core i7@3.7 GHz with 4 cores and Hyper-threading enabled.

Benchmark	**Time with Valgrind**	**Time with Nullgrind**	**Slowdown**
Bzip2	5 mins 44 secs	1 min 43 secs	2.33
Gobmk	1 sec 253 ms	543 ms	1.30
Astar	1 min 29 sec	28 sec	2.17
Lbm	32 secs	7 secs	3.57
H264ref	5 min 35 secs	57 secs	4.18

To calculate the slowdown, the benchmark binaries were first run with memcheck tool and then with nullgrind. Since, Nullgrind doesn't do any instrumentation; there is no slowdown due to instrumentation involved.

We also tested Mozilla Firefox binary against Valgrind's memcheck plug-in and the overhead came to be 5.17.

8. References

1. http://en.wikipedia.org/wiki/Memory_leak
2. 'Using Valgrind to detect undefined value errors with bit precision'. Nicholas Nethercote, Julian Seward.
3. 'How to Shadow every byte of memory used by a program'. Nicholas Nethercote, Julian Seward.
4. http://www.valgrind.org/
5. http://linux.die.net/man/3/efence
6. http://www.splint.org/manual/html/sec5.html
7. http://lwn.net/Articles/187193/
8. Precise detection of memory leaks' by Jonas Maebe Michiel Ronsse Koen De Bosschere, Ghent University, ELIS Department.
9. ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/unixsuites/pdf/purify/purify.pdf