Verification & Testing
Memory Debuggers

Roderick Bloem

V&T 3

Who uses valgrind? Why?

Memory Problems

```
#define intmalloc(n)
          (int*)malloc(n*sizeof(int));
```

Uninitialized read

```
int a, b;
a = b;
```

Unallocated read

```
int *p = intmalloc(4);
printf(“%d”, p[4]);
```

Unallocated write

```
int *p = intmalloc(3);
p[3] = 10;
```

Write after free

```
int *p = intmalloc(4);
free(p);
p[2] = 10;
```

Memory Leak

```
int *p = intmalloc(4);
end of program
```

Freeing unallocated memory

```
int *p;
free(p);
```

or

```
p = intmalloc(10);
free(p);
free(p);
```

Are these real problems?

None of These Errors Dump Core

- These errors do not always dump core. (Depending on compiler, OS)
    - sometimes produces expected, sometimes unexpected results
- Uninitialized read: results depend on execution
    - `int a, b;`
    - `a = b;`
- Unallocated read. Returns data from different data structure.

```
int *p = malloc(4*sizeof(int));
int b;
b = p[4];
```

- Unallocated write may overwrite other data. May dump core if p points to the end of an allocated page,

```
int *p = intmalloc(3);
p[3] = 10;
```

- Write after free: may overwrite other data if memory is reallocated before write. May dump core if memory is returned to OS

```
int *p = intmalloc(4);
free(p);
p[2] = 10;
```

- Memory Leak. Slows program down and may dump core if in a loop.

```
int *p = malloc(4*sizeof(int));
end of program
```

- Freeing unallocated memory. May lead to a core dump, may not.

```
int *p;
free(p);
```

or

```
p = intmalloc(10);
free(p);
free(p);
```

Memory Errors

Memory Errors are

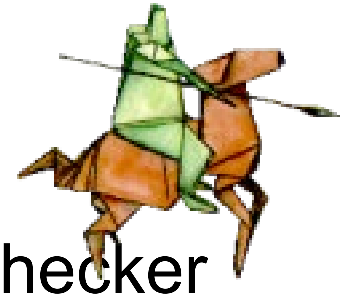
- hard to find
- often show themselves only occasionally
- often become apparent in different piece of code
- happen frequently!

Finding Memory Errors

List of tools that help with memory errors:

- IBM’s Purify (Rational)
- Valgrind (open source, Linux)
- electric fence (open source)
- dmalloc (open source)
- Clang & gcc sanitizer
- Intel parallel inspector
- Google’s AddressSanitizer
- … (see Wikipedia Memory Debugger)

Valgrind



Valgrind: suite of tools, including a memory checker

- Translate to intermediate code
- Instrument intermediate code
- Execute on virtual CPU

Memcheck:

increases code size 12x.

Runs 25-50x slower.

Null:

adds nothing,

runs 4x slower

Valgrind Workings

Add ***shadow memory***

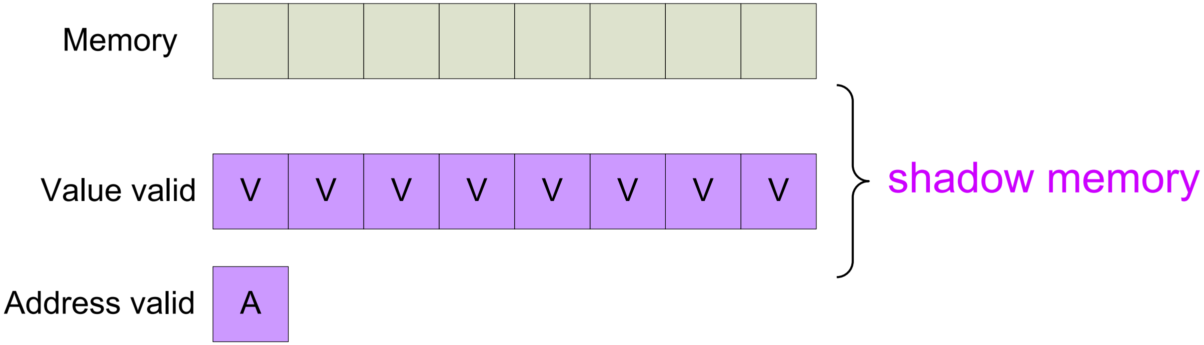
Per byte of memory add

- **V-bits:** One byte to store whether each bit has a valid value
- **A-bit:** One bit to store whether byte has been allocated

We want to find

- accesses where memory is not allocated
- decisions that depend on uninitialized values.

Uninitialized copies are OK!



Memory

Value valid V V V V V V V V

Address valid A

shadow memory

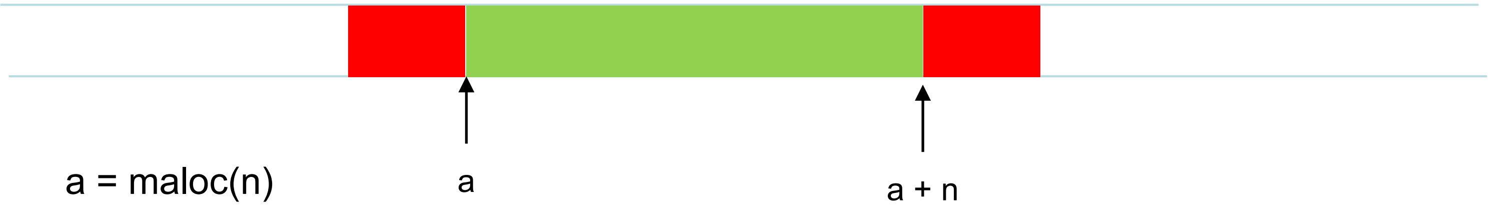
Valgrind Workings

- **Read or write:** Check A-bit
- **Write:** set V-bits
- **Load register from memory:** also load value bits into shadow register
- **Store register to memory:** store value bits into shadow memory
- **Value is used as address:** check V-bits
- **Branch depends on values:** check V-bits

When value bits have been checked, they are set (prevents same error from being reported again)

Malloc and free

- **Malloc/new:** address is valid, value is not. Keep "red zone" (address bits set to false) between memory chunks
- **free/delete:** check that memory has been allocated, prevent memory from being reallocated for as long as possible. Set A-bit to 0.



Examples

0	A *a	V *a	A *b	V *b
`main(){`				
`  int* a = malloc(sizeof(int));`				
`  int* b = malloc(sizeof(int));`				
`  *b = 3;`				
`  *b = *a;`				
`}`				

I will assume 16-bit basic data types
I will write k^n for a bitstring consisting of ks

Examples

0	A *a	V *a	A *b	V *b
main(){	0000	0^16	0000	0^16
int* a = malloc(sizeof(int));	1111	0^16	0000	0^16
int* b = malloc(sizeof(int));	1111	0^16	1111	0^16
*b = 3;	1111	0^16	1111	1^16
*b = *a;	1111	0^16	0000	0^16
}				

I will assume 32-bit basic data types
I will write k^n for a bitstring consisting of ks

Examples

	A *a	V *a
`main(){`		
`   int* a;`		
`   *a = *a & 0xfffe;`		
`}`		

Examples

	A *a	V *a
main(){		
int* a;	0000	0^16
*a = *a & 0xfffe;	0000	0000 0000 0000 0001
}		

Examples

	A *p	V *p
int *p;		
int x = 1;		
p = malloc(sizeof(int));		
if(x){		
*p = 3;		
free(p);		
printf("%d",*p);		
} else		
printf("%d",*p);		

Examples

	A *p	V *p
int *p;	0000	0^16
int x = 1;	0000	0^16
p = malloc(sizeof(int));	1111	0^16
if(x){	1111	0^16
*p = 3;	1111	1^16
free(p);	0000	1^16
printf("%d",*p);	**WARNING: ACCESS TO UNALLOCATED VALUE** 0000	1^16
} else		
printf("%d",*p);		

Examples

	A *p	V *p
`int *p;`	0000	0^16
`int x = 0;`		
`p = malloc(sizeof(int));`		
`if(x){`		
`  *p = 3;`		
`  free(p);`		
`  printf("%d",*p);`		
`} else`		
`  printf("%d",*p);`		

Examples

	A *p	V *p
int *p;	0000	0^16
int x = **0**;	0000	0^16
p = malloc(sizeof(int));	1111	0^16
if(x){		
*p = 3;		
free(p);		
printf("%d",*p);		
} else	1111	0^16
printf("%d",*p);	**WARNING: ACCESS TO UNINITIALIZED VALUE** 0000	0^16

More Details

Validity is kept on bit level. Need to properly handle

- Bit operations such as AND and OR
  - $? \wedge 0 = 0$, but $? \wedge 1 = ?$
  - $? \vee 0 = ?$, but $? \vee 1 = 1$
- Additions
- Shifts
- a XOR a
- Etc…

Example: Uninitialized Copy

```
int *p, *q;
max = user input, < 1024

p = (int*) malloc(1024*sizeof(int));
q = (int*) malloc(1024*sizeof(int));

for(i = 0; i < max; i++)
  p[i] = 0;

memcpy(q, p, 1024 * sizeof(int));

for(i = 0; i < max; i++)
  if(q[i])
    printf(“strange!\n”);

free(p); free(q);
```

This program is deemed correct by valgrind. Note that uninitialized values may be copied, as long as they are not visible.

Another example: a struct with four allocated bytes often takes up 8 bytes. Copying the struct copies uninitialized memory.

Bugs Valgrind Cannot Catch

```
void f(){
  int a[10];
  int b[10];

  printf(“%d\n”,b[0]);
  a[10] = 5;
  printf(“%d\n”,b[0]);
}
```

Valgrind cannot catch buffer overflows on static and local data. (only on malloc’ed data.) (*Why?)*

In compiled code, there is no difference between a[10] and b[0]

```
Valgrind --tool=memcheck --leak-check=yes
--suppressions=suppress.supp
```

Bugs Valgrind Cannot Catch

```
void f(){
  int a[10];
  int b[10];

  printf(“%d\n”,b[0]);
  a[10] = 5;
  printf(“%d\n”,b[0]);
}
```

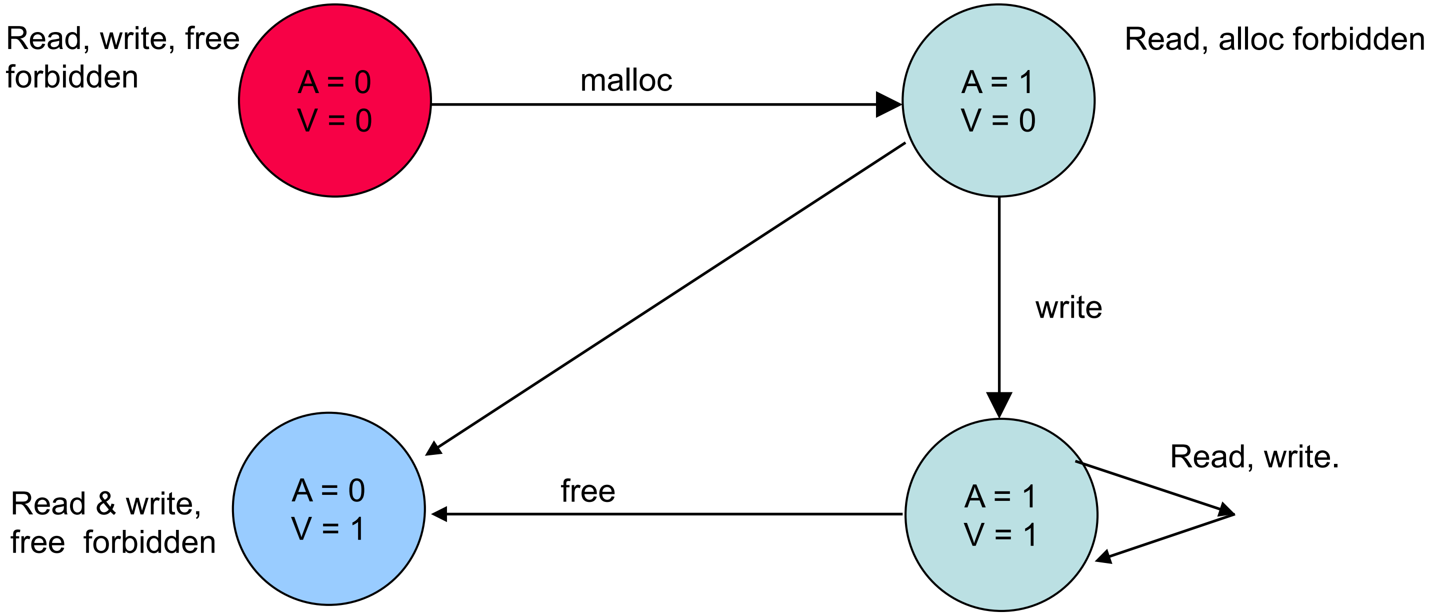
Valgrind cannot catch buffer overflows on static and local data. (only on malloc’ed data.) (*Why?)*

```
Valgrind --tool=memcheck --leak-check=yes
--suppressions=suppress.supp
```

Purify

Purify uses two bits of status per byte of memory

- Valid address?
- Valid data?



Purify

Less memory overhead: per byte, not per bit

No virtual CPU

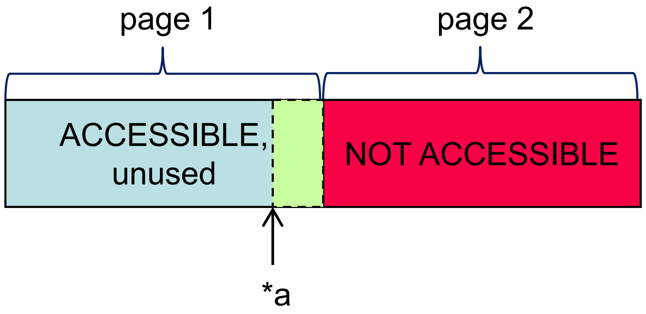
- Error flagged when uninitialized bytes read: uninitialized copies not allowed
- Faster, but more spurious warnings

Electric Fence

Memory is divided into *pages* (4096 bytes, usually)

- For every malloc, adjacent page of inaccessible memory is allocated
- MMU checks accesses to inaccessible pages without time overhead
- Memory overhead: every datastructure is at least 1 page
  - Big overhead if you have small datastructures!
  - The inaccessible page does not really count
- No virtual CPU, no annotation
- Only catches index too large accesses

```
a = malloc(128*sizeof(int))
```



More Valgrind Tools

Valgrind also includes

- Helgrind & Data Race Detector implement race condition detection (‘happens-before’)
- Massif is a heap profiler
- Callgrind is a profiler
- Cachegrind analyzes cache usage
- AddrCheck uses only A bits
- NullGrind