cse504 class presentation

- LCLint (PLDI’96 paper)
- Splint (IEEE’02 paper)
- Prefix (Intrinsa SP&E’00 paper)

jaeyeon.jung@intel.com
04/07/2010
part I

static detection of dynamic memory errors
the problem

• memory errors are hard to detect at compile-time

• observations
  – many bugs result from invalid assumptions about the results of functions and the values of parameters and global variables.
  – these bugs are platform independent.
memory errors

• misuses of null pointers
• lack of memory allocation or deallocation
• uses of undefined storage
• unexpected aliasing
extern char *gname;

void setName (char *pname) {
    gname = pname;
}

extern char *gname;

void setName (char *pname) {
    gname = pname;
}

1. must not be a sole ref.
sample.c

extern char *gname;

void setName (char *pname) {
    gname = pname;
}

2. gname and pname are aliased.
sample.c

extern char *gname;

void setName (char *pname) {
    gname = pname;
}

3. gname may not be dereferenced if pname is a null pointer.
sample.c

extern char *gname;

void setName (char *pname) {
    gname = pname;
}

4. gname may not be dereferenced as a rvalue unless pname pointed to defined storage.
the approach

• make assumptions explicit with annotations
  – function interfaces, variables, types
• extend LCLint to statically detect the errors
  – LCLint became secure programming Lint
  http://www.splint.org/
annotations

• syntactic comments
  – e.g., /* @null@ */

• used in
  – type declaration
  – function parameter or return value declarations
  – global and static variable declarations
annotations --- null pointers

1 extern char *gname;

2

3 void setName (/*@null@*/ char *pname) {
4     gname = pname;
5 }

sample.c:5: function returns with non-null global gname referencing null storage.
sample.c:4: storage gname may become null.
annotations --- null pointers

extern char *gname;
extern /*@truenull@*/
    isNull (/*@null@*/ char *x);
void setName (/*@null@*/ char *
pname) {
    If (!isNull(pname)) {
        gname = pname;
    }
}
annotations --- definition

- out: referenced storage need not be defined
- in/partial/undef: referenced storage is completely/partially/not defined
- reldef: value assumed to be defined when it is used, but need not be assigned to defined storage
extern /*@only@*/ char *gname;

void setName (/*@temp@*/ char *pname) {
    gname = pname;
}

1. memory leak
2. gname will become a dead pointer if the caller deallocates the actual parameter
annotations --- aliasing

- unique: parameter aliasing
- returned: a reference to the parameter may be returned
evaluation --- toy program

- employee database program (1K LoC)
- adding annotations is an iterative process
  - 13 only, 1 out, 1 null
- found three bugs
  - null pointers, allocation, aliasing
evaluation --- toy program

```c
typedef/*@null*/ struct _list
{
  /*@only*/ char *this;
  /*@null*//*@only*/ struct _list *next;
} *list;

extern/*@out*//*@only*/ void *
  smalloc(size_t);

void
list_addh(/*@temp*/ list l,
  /*@only*/ char *e)
{
  if (l != NULL)
    {
      while (l->next != NULL)
        {
          l = l->next;
        }
      l->next = (list)
        smalloc(sizeof(*l->next));
      l->next->this = e;
    }
}
```

evaluation --- LCLint

- 100K lines of code
- < 4 minutes to check
- adding all annotations required a few days over the course of a few weeks by one person
- revealed limitations of strict annotations – e.g., handling an error condition
summary

• the annotations improve
  – static checking
  – maintaining and developing code

• a combination of static checking and run-time checking is promising to producing reliable code.
Improving security using extensible lightweight static analysis
the problem

• the techniques for avoiding security vulnerabilities are not codified into the software development process

• C is difficult to secure
  – unsafe functions
  – confusing APIs
the solution

• Splint: a lightweight static analysis tool for ANSI C
  – detects stack and heap-based buffer overflow vulnerabilities
  – support user-defined checks
    • constrain the values of attributes at interface points
    • specify how attributes change
the challenges

• false positive & false negatives
• tradeoff between precision and scalability
  – limited to data flow analysis within procedure bodies
  – merges possible paths at branch points
  – use heuristics to analyze loop
example --- buffer overflow analysis

• requires, ensures
• maxSet
  – highest index that can be safely written to
• maxRead
  – highest index that can be safely read
• char buffer[100];
  – ensures maxSet(buffer) == 99
SecurityFocus.com Example

```c
char *strncat (char *s1, char *s2, size_t n)
/*@requires \text{maxSet(s1)} 
\geq \text{maxRead(s1)} + n*/

void func(char *str){
    char buffer[256];
    strncat(buffer, str, sizeof(buffer) - 1);
    return;
}
```

Source: Secure Programming working document, SecurityFocus.com

http://www.cs.virginia.edu/evans/talks/usenix.ppt
Warning Reported

char * strncat (char *s1, char *s2, size_t n)
/*@requires maxSet(s1) >= maxRead(s1) + n @*/
char buffer[256];
strncat(buffer, str, sizeof(buffer) - 1);

strncat.c:4:21: Possible out-of-bounds store:
   strncat(buffer, str, sizeof((buffer)) - 1);
Unable to resolve constraint:
   requires maxRead (buffer @ strncat.c:4:29) <= 0
needed to satisfy precondition:
   requires maxSet (buffer @ strncat.c:4:29)
     >= maxRead (buffer @ strncat.c:4:29) + 255
derived from strncat precondition:
   requires maxSet (<parameter 1>)
     >=  maxRead (<parameter1>) + <parameter 3>

http://www.cs.virginia.edu/evans/talks/usenix.ppt
example --- taint analysis

```
attribute taintedness
  context reference char *
  oneof untainted, tainted
annotations
  tainted reference ==> tainted
  untainted reference ==> untainted
transfers
  tainted as untainted ==> error "Possibly tainted storage used as untainted."
merge
  tainted + untainted ==> tainted
defaults
  reference ==> tainted
  literal ==> untainted
  null ==> untainted
end
```

example --- taint analysis

char *strcat
  (/*@returned@*/ char *s1,
   char *s2)
/*@ensures s1:taintedness =
   s1:taintedness | s2.taintedness@*/

annotated declarations define taint propagation at the
interface for standard library functions
evaluation --- wu-ftp

- 20K LoC
- < 4 seconds to check the code on a slow (1.2GHz) machine
- found a few known bugs using the taint analysis
- 101 warnings after adding 66 annotations
  - 76 false positives
    - external assumptions, arithmetic limitations, alias analysis, flow control, loop heuristics
wu-ftpnd vulnerability

```c
int access_ok( int msgcode) {
    char class[1024], msgfile[200];
    int limit;

    ...

    limit = acl_getlimit(class, msgfile);
}
```

http://www.cs.virginia.edu/evans/talks/usenix.ppt
summary

• static analysis is promising but
  – limited to finding problems that manifest as inconsistencies between the code and assumptions documented in annotations
  – annotating legacy code is laborious

• static analysis helps codifying knowledge into tools not to avoid making same mistakes
A static analyzer for finding dynamic programming errors
the problem

• many bugs are caused by the interaction of multiple functions and may be revealed only in unusual cases
  – compilers, Lint are limited to intra-procedural checks
  – annotation checkers require too much work
  – debugging tools incur performance overhead
the design goals

• practical
  – effectively check C/C++ programs
  – leverage information automatically derived from the program text

• analysis limited to achievable paths

• actionable
  – automatic characterization of defects
PREfix’s key concept

• simulate functions using VM
  – achievable paths
• automatically generate a function’s model
• bottom-up analysis
PREfix

• parse the source code into abstract syntax tree
• run topological sort for simulating functions from the leaf
• load existing models for relevant functions
• simulate functions
  – simulate achievable paths
  – per-path simulation
per-path simulation

• memory: exact values and predicates
  – known exact value, initialized but unknown value, uninitialized value
  – dereference
• operations on memory
  – setting, testing, assuming
• conditions, assumptions and choice points
• end-of-path analysis
  – leak analysis
model -- deref

```c
1 int deref(int *p)
2 {
3     if (p==NULL)
4         return NULL;
5     return *p;
6 }
```
model -- deref

```c
int deref(int *p)
{
    if (p==NULL)
        return NULL;
    return *p;
}
```

```c
(deref
  (param p)
  (alternate return_0
    (guard peq p NULL)
    (constraint memory_initialized p)
    (result peq return NULL)
  )
  (alternate return_X
    (guard pne p NULL)
    (constraint memory_initialized p)
    (constraint memory_valid_pointer p)
    (constraint memory_initialized *p)
    (result peq return *p)
  )
)
```

model generation

• record all the per-path memory state
  – tests -> constraints
• save externally visible states
  – parameters, return values and globals
• merge states
  – for performance
  – equivalent merging (e.g., one assumes $x>0$ and the other assumes $x<=0$)
  – no aggressive merging (e.g., [merge *p=5 and *p=8 -> *p is initialized] caused accuracy issues
evaluation

OK performance on a slow machine

<table>
<thead>
<tr>
<th>Program</th>
<th>Language</th>
<th>Number of files</th>
<th>Number of lines</th>
<th>PREfix parse time</th>
<th>PREfix simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla</td>
<td>C++</td>
<td>603</td>
<td>540 613</td>
<td>2 h 28 min</td>
<td>8 h 27 min</td>
</tr>
<tr>
<td>Apache</td>
<td>C</td>
<td>69</td>
<td>48 393</td>
<td>6 min</td>
<td>9 min</td>
</tr>
<tr>
<td>GDI Demo</td>
<td>C</td>
<td>9</td>
<td>2 655</td>
<td>1 s</td>
<td>15 s</td>
</tr>
</tbody>
</table>
Table II. Warnings reported in sample public domain software.

<table>
<thead>
<tr>
<th>Warning</th>
<th>Mozilla</th>
<th>Apache</th>
<th>GDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using uninitialized memory</td>
<td>26.14%</td>
<td>45%</td>
<td>69%</td>
</tr>
<tr>
<td>Dereferencing uninitialized pointer</td>
<td>1.73%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dereferencing NULL pointer</td>
<td>58.93%</td>
<td>50%</td>
<td>15%</td>
</tr>
<tr>
<td>Dereferencing invalid pointer</td>
<td>0</td>
<td>5%</td>
<td>0</td>
</tr>
<tr>
<td>Dereferencing pointer to freed memory</td>
<td>1.98%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Leaking memory</td>
<td>9.75%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Leaking a resource (such as a file)</td>
<td>0.09%</td>
<td>0</td>
<td>8%</td>
</tr>
<tr>
<td>Returning pointer to local stack variable</td>
<td>0.52%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Returning pointer to freed memory</td>
<td>0.09%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Resource in invalid state</td>
<td>0</td>
<td>0</td>
<td>8%</td>
</tr>
<tr>
<td>Illegal value passed to function</td>
<td>0.43%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Divide by zero</td>
<td>0.35%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total number of warnings</td>
<td>1159</td>
<td>20</td>
<td>13</td>
</tr>
</tbody>
</table>
The decrease in coverage as more models are introduced

<table>
<thead>
<tr>
<th>Model set</th>
<th>Execution time (minutes)</th>
<th>Statement coverage</th>
<th>Branch coverage</th>
<th>Predicate coverage</th>
<th>Total warning count</th>
<th>Using uninit memory</th>
<th>NULL pointer deref</th>
<th>Memory leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>12</td>
<td>90.1%</td>
<td>87.8%</td>
<td>83.9%</td>
<td>15</td>
<td>2</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>System</td>
<td>13</td>
<td>88.9%</td>
<td>86.3%</td>
<td>82.1%</td>
<td>25</td>
<td>6</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>System &amp; auto</td>
<td>23</td>
<td>73.1%</td>
<td>73.1%</td>
<td>68.6%</td>
<td>248</td>
<td>110</td>
<td>24</td>
<td>124</td>
</tr>
</tbody>
</table>
summary

• PREfix is a dynamic checker with
  – adjustable thresholds on path coverage
  – heuristics to manage paths to check
  – efficient function models

• bugs found by PREfix
  – caused by multi-function interactions
  – off main code paths
  – more found in younger code
take-away

• LCLint (PLDI paper) & Splint (IEEE paper)
  – static analysis with annotations
  – manual, iterative process but improves maintaining and developing code

• Prefix (Intrinsa SP&E paper)
  – dynamic checker with models and heuristics
  – automatic, inter-procedural analysis, but may produce lots of false positives