The Importance of Benchmarks for Tools that Find or Prevent Buffer Overflows

Richard Lippmann, Michael Zhivich Kendra Kratkiewicz, Tim Leek, Graham Baker, Robert Cunningham

MIT Lincoln Laboratory
lippmann@ll.mit.edu

To be presented at the Workshop on the Evaluation of Software Defect Detection Tools, Co-located with the PLDI 2005 Conference, Chicago 12 June 2005

*This work was sponsored by the Advanced Research and Development Activity under Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Our Experience with Buffer Overflow Detection Tools – Benchmarks are Essential

- **An initial literature review led us to believe that tools could reliably find buffer overflows**

BOON

Splint

PolySpace TECHNOLOGIES

Automatic Detection of Run-Time Errors at Compile Time

A Unique Solution To Reduce Software Testing Costs

Ensuring Flawless Software Reliability

- **We created a hierarchy of buffer overflow benchmarks**
  1. **Large full programs**
     - **Historic versions of BIND, Sendmail, WU-FTP servers with known buffer-overflow vulnerabilities (14)**
     - **Recent versions of gzip, tar, OpenSSL, Apache**
  2. **14 Model Programs extracted from servers with known buffer-overflow vulnerabilities (169-1531 lines of code each)**

     Available from http://www.ll.mit.edu/IST/corpora.html
  3. **291 Small Diagnostic C Test Cases**
     - **Created using a buffer overflow taxonomy with 22 attributes, each case varies one attribute**

     Available from Kendra Kratkiewicz, kendra@ll.mit.edu

Model Program Excerpt for Sendmail GECOS Overflow CVE-1999-0131

```
ADDRESS *recipient(...) {
  ...
  else {
  /* buffer created */
    char nbuf[MAXNAME + 1];
    buildfname(pw->pw_gecos,
            pw->pw_name, nbuf);
    ...
  }
}
```

```
void buildfname(gecos, login, buf)
  register char *gecos;
  char *login;
  char *buf; {
  ...
  register char *bp = buf;
  /* fill in buffer */
  for (p = gecos; *p != '\0' &&
                  *p != ',' &&
                  *p != ';' &&
                  *p != '%'; p++) {
    if (*p == '&') {
      /* BAD */
      (void) strcpy(bp, login);
      *bp = toupper(*bp);
      while (*bp != '\0')
       bp++;
    }
    else
      /* BAD */
      *bp++ = *p;
  }
  /* BAD */
  *bp = '\0';
}
```

Diagnostic C Test Case Taxonomy

Taxonomy Attributes

Attribute Number	Attribute Name
1	Write/Read
2	Upper/Lower Bound
3	Data Type
4	Memory Location
5	Scope
6	Container
7	Pointer
8	Index Complexity
9	Address Complexity
10	Length/Limit Complexity
11	Alias of Buffer Address
12	Alias of Buffer Index
13	Local Control Flow
14	Secondary Control Flow
15	Loop Structure
16	Loop Complexity
17	Asynchrony
18	Taint
19	Runtime Environment Dependence
20	Magnitude
21	Continuous/Discrete
22	Signed/Unsigned Mismatch

Scope

Value	Description
0	same
1	inter-procedural
2	global
3	inter-file/inter-procedural
4	inter-file/global

Magnitude

Value	Description	Example
0	none	buf[9] = 'A';
1	1 byte	buf[10] = 'A';
2	8 bytes	buf[17] = 'A';
3	4096 bytes	buf[4105] = 'A';

OK and BAD (Vulnerable) Diagnostic C Test Case Example

OK Test Case

```
/* Taxonomy Classification: 00010000000000000000
* WRITE/READ 0 write
* WHICH BOUND 0 upper
* DATA TYPE 0 char
* MEMORY LOCATION 1 heap
* SCOPE 0 same
* CONTAINER 0 no
* POINTER 0 no
* INDEX COMPLEXITY 0 constant
* ADDRESS COMPLEXITY 0 constant
* LENGTH COMPLEXITY 0 N/A
* ADDRESS ALIAS 0 none
* INDEX ALIAS 0 none
* LOCAL CONTROL FLOW 0 none
* SECONDARY CONTROL FLOW 0 none
* LOOP STRUCTURE 0 no
* LOOP COMPLEXITY 0 N/A
* ASYNCHRONY 0 no
* TAINT 0 no
* RUNTIME ENV. DEPENDENCE 0 no
* MAGNITUDE 0 no overflow
* CONTINUOUS/DISCRETE 0 discrete
* SIGNEDNESS 0 no
*/
#include <stdlib.h>
#include <assert.h>
int main(int argc, char *argv[])
{
char * buf;
buf=(char *)malloc(10*sizeof(char));
assert (buf != NULL);

/* OK */
buf[9] = 'A';

return 0;}
```

BAD (Vulnerable) Test Case

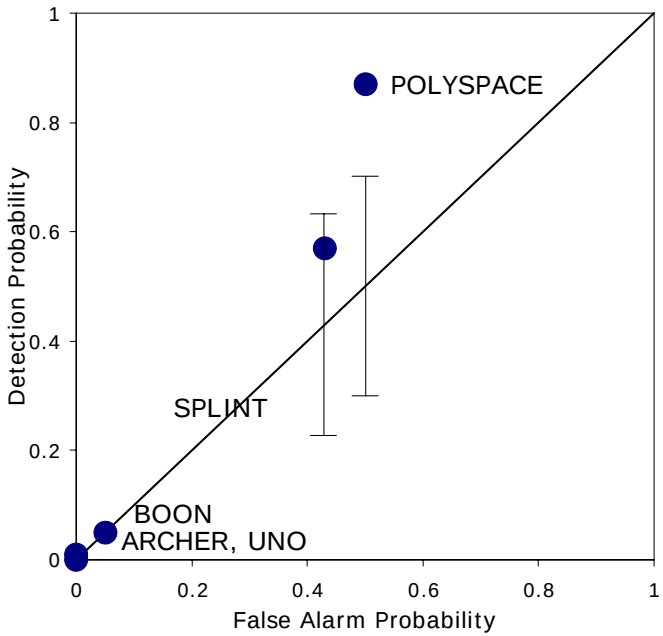
```
/* Taxonomy Classification: 00010000000000000100
* WRITE/READ 0 write
* WHICH BOUND 0 upper
* DATA TYPE 0 char
* MEMORY LOCATION 1 heap
* SCOPE 0 same
* CONTAINER 0 no
* POINTER 0 no
* INDEX COMPLEXITY 0 constant
* ADDRESS COMPLEXITY 0 constant
* LENGTH COMPLEXITY 0 N/A
* ADDRESS ALIAS 0 none
* INDEX ALIAS 0 none
* LOCAL CONTROL FLOW 0 none
* SECONDARY CONTROL FLOW 0 none
* LOOP STRUCTURE 0 no
* LOOP COMPLEXITY 0 N/A
* ASYNCHRONY 0 no
* TAINT 0 no
* RUNTIME ENV. DEPENDENCE 0 no
* MAGNITUDE 1 1 byte
* CONTINUOUS/DISCRETE 0 discrete
* SIGNEDNESS 0 no
*/
#include <stdlib.h>
#include <assert.h>
int main(int argc, char *argv[])
{
char * buf;
buf=(char *)malloc(10*sizeof(char));
assert (buf != NULL);

/* BAD */
buf[10] = 'A';

return 0;}
```

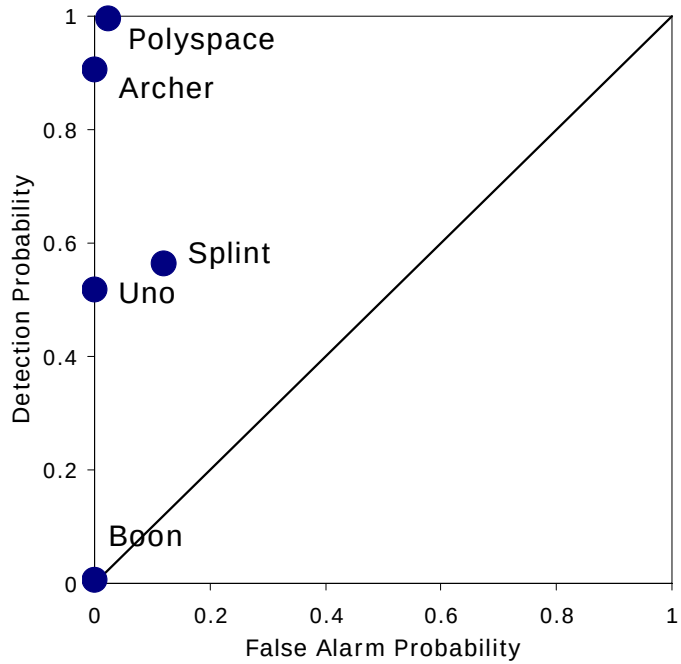
Evaluating Static Analysis Tools with Model Programs and Test Cases

14 MODEL PROGRAMS



- Most tools can't handle real server code!
- They also exhibit poor performance on extracted model programs
  - Low detection and high false alarm rates
  - Only Polyspace is better than guessing

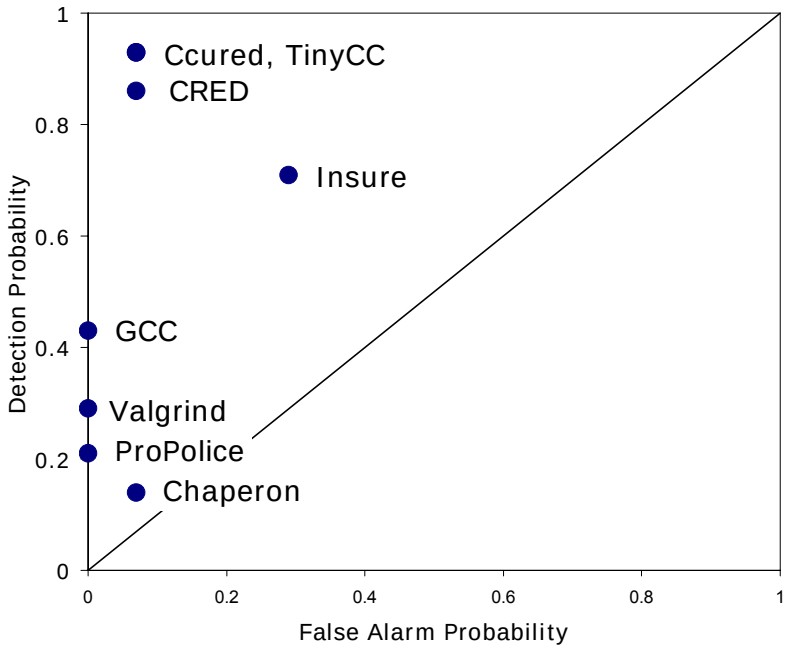
291 Diagnostic Test Cases



- Good performance for Archer and Polyspace on simple test cases but
  - Run time for Polyspace is more than two days
  - Archer doesn't perform inter-procedural analysis or handle string functions

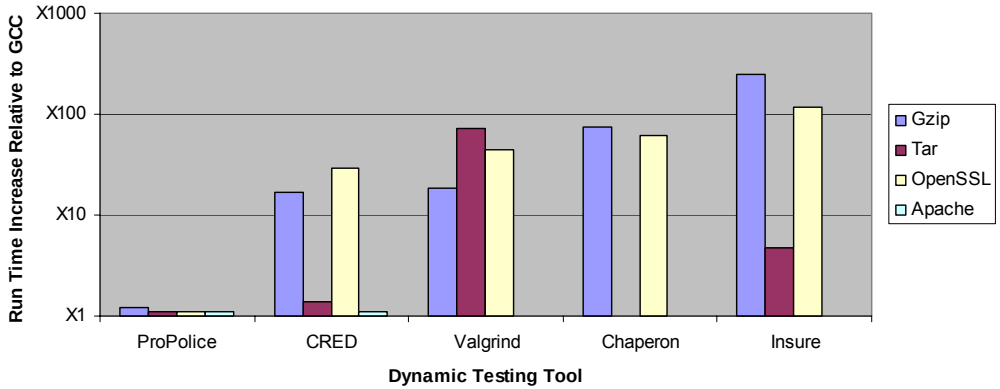
Evaluating Dynamic Test Instrumentation Tools with Benchmarks

14 MODEL PROGRAMS



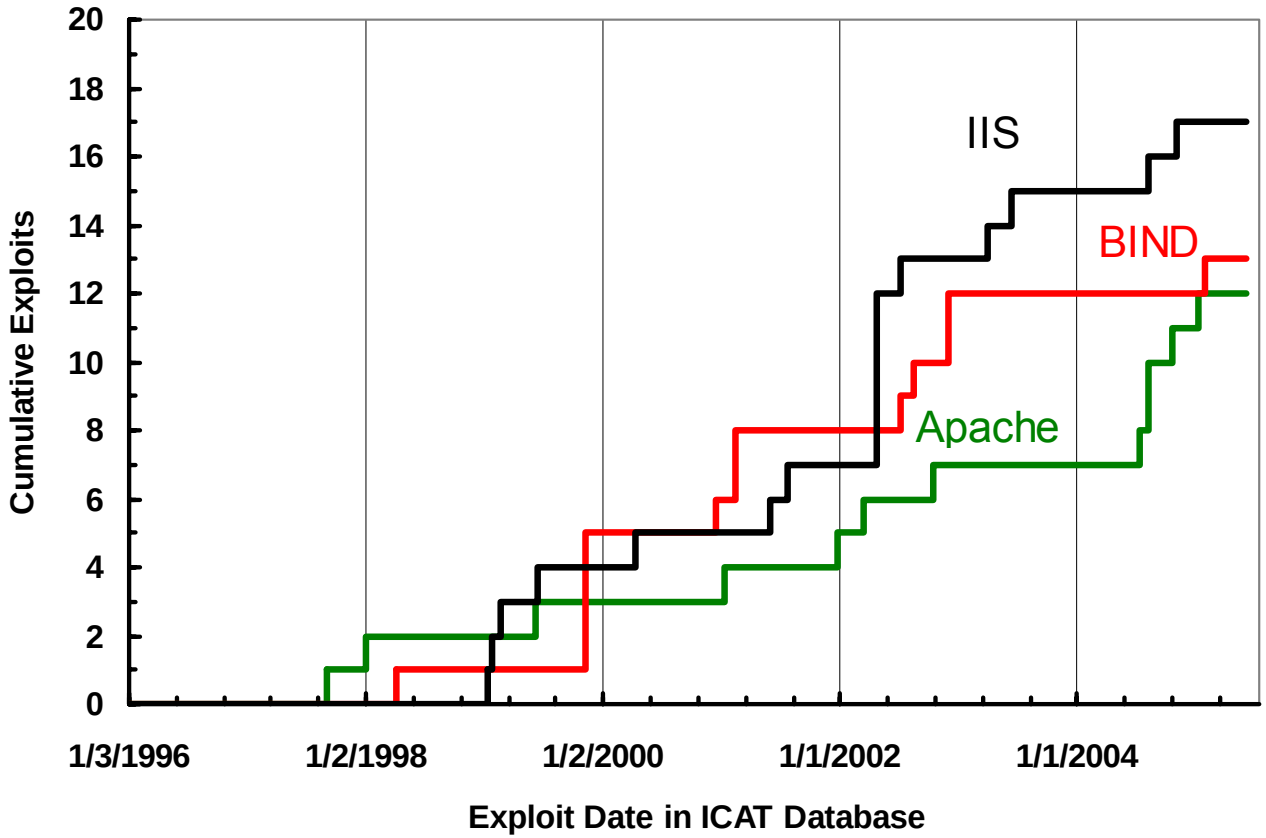
- Some tools accurately detect most overflows in model programs
  - CCured, TinyCC, CRED
  - Misses are caused by errors in implementation or limited analyses

Increase in Run Time Compared to GCC



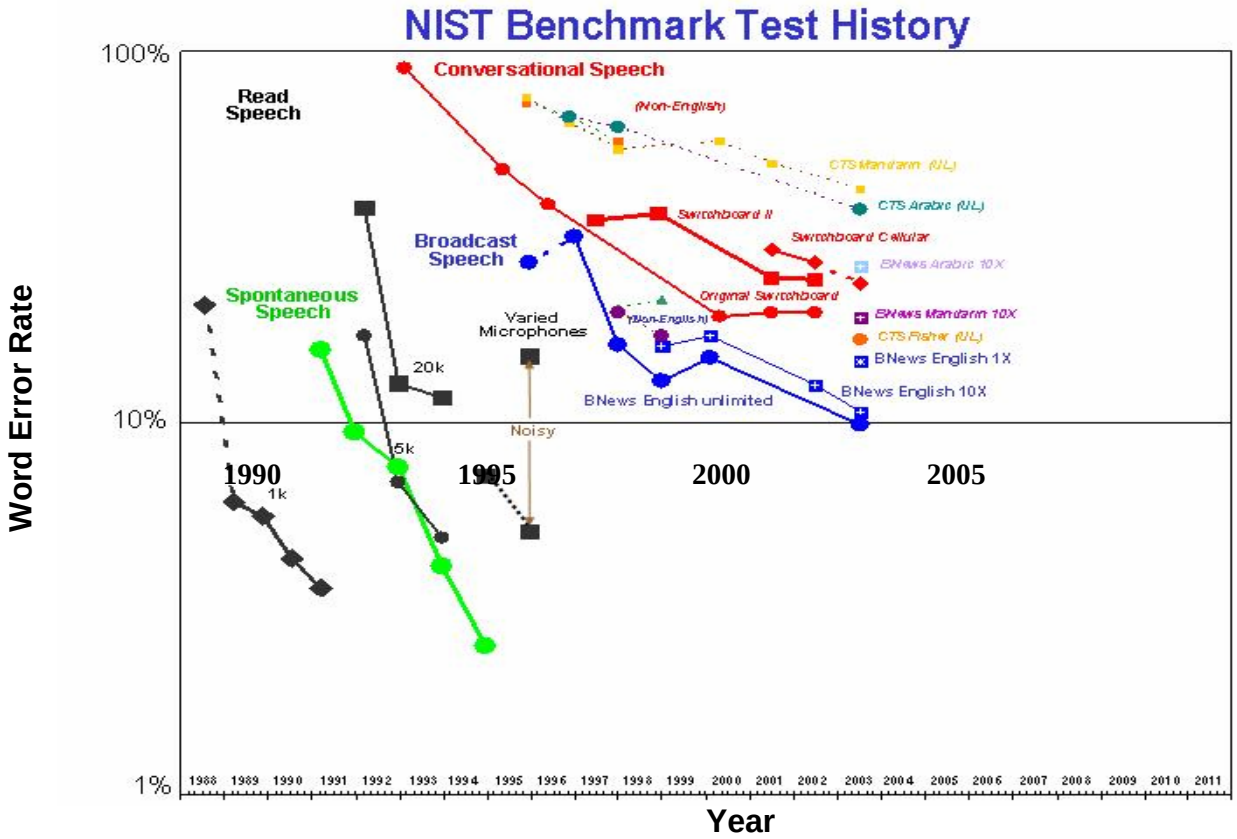
- Some tools can't compile large programs (e.g. CCured, TinyCC,)
- Some tools exhibit excessive (x100) increases in run time (e.g. Chaperon, Insure)
- Only CRED combines good detection with reasonable run times.

Why Do Remotely Exploitable Buffer Overflows Still Exist?



- As many new buffer overflow vulnerabilities are being found each year today in important internet software as were being found six years ago

Speech Recognition Benchmarks Led to Dramatic Performance Improvements



- 1969 – Mad inventors and untrustworthy engineers, no progress, work has been an experience with no knowledge gained (Pierce, 1969)
- 1981 – First publicly available speech data base (Doddington, 1981)
- Today – Dramatic progress and many deployed speech recognizers, major focus on corpora and benchmarks (Pallet, 2004)

Comments

- Don't shoot the messenger
  - It is essential to benchmark tool performance
  - How else can you know how well an approach works and set expectations for tool users?
  - How else can you obtain diagnostic information that can be used to guide further improvements?

- Benchmarks should be fair, comprehensive and appropriate
  - Provide ground truth, measure detection and false alarm rates, run times, memory requirements, …
  - Include tasks appropriate for the tool being evaluated

- Using tools that "find hundreds of bugs on …" may be detrimental because they provide a false sense of security
  - What are their detection and miss rates?
  - Are these the type of bugs that we really care about?

- Developers have to think more about how tools fit into the code development/use lifecycle

References

- Doddington, G. R. and T. B. Schalk (1981). Speech Recognition: Turning Theory into Practice,. IEEE Spectrum,**:** 26-32.
- Kratkiewicz, K. J. and R. Lippmann (2005). Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools. Workshop on the Evaluation of Software Defect Detection Tools.
- Kratkiewicz, K. J. (2005). Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code. ALM in IT Thesis in the Harvard University Extension Program.
- Pallett, D. S. (2003). A Look at NIST's Benchmark ASR Tests: Past, Present, and Future, http://www.nist.gov/speech/history/pdf/NIST_benchmark_ASRtests_2003.pdf
- Pierce, J. (1970). “Whither speech recognition?” Journal of the Acoustical Society of America **47**(6): 1616-1617.
- Zhivich, M., T. Leek, et al. (2005). Dynamic Buffer Overflow Detection. Workshop on the Evaluation of Software Defect Detection Tools.
- Zitser, M., R. P. Lippmann, et al. (2004). Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code. Proceedings ACM Sigsoft 2004/FSE Foundations of Software Engineering Conference, http://www.ll.mit.edu/IST/pubs/04_TestingStatic_Zitser.pdf.