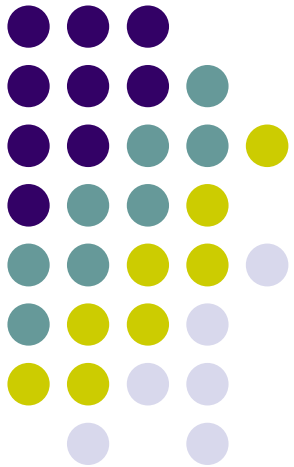
Finding Security Violations by Using Precise Source-level Analysis



by

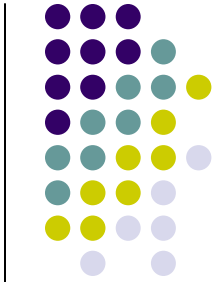
V.Benjamin Livshits and Monica Lam

{livshits, lam}@cs.stanford.edu

SUIF Group

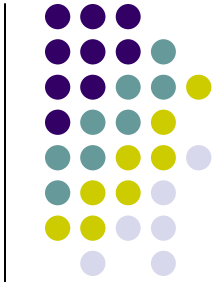
CSL, Stanford University

Computer Break-ins: Major problem



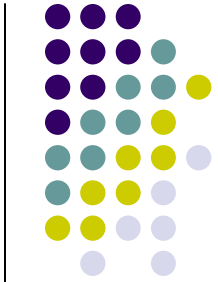
- Software break-ins: *relatively* easy to do: a lot of prior art
- An article selection from [destroy.net]:
  - Smashing The Stack For Fun And Profit [Aleph One]
  - How to write Buffer Overflows [Mudge]
  - Finding and exploiting programs with buffer overflows [Prym]
- Sites like that describe techniques and provide tools to simplify creating new exploits

Potential Targets



- Typical targets:
  - Widely available UNIX programs: sendmail, BIND, etc.
  - Various server-type programs
    - ftp, http
    - pop, imap
    - irc, whois, finger
  - Mail clients (overrun filenames for attachments)
    - Netscape mail (7/1998)
    - MS Outlook mail (11/1998)
  - The list goes on and on…

Sad Consequences



- Patching mode: need to apply patches in a timely manner
- Recent cost estimate: a survey by analyst group Baroudi Bloor [www.baroudi.com]
  1. Lost Revenue due to Down Time – biggest cost

  …but also
  2. System Admin Time Costs
  3. Development Costs
  4. Reputation and Good Will -- cannot be measured

Baroudi Bloor report: failure to patch on time

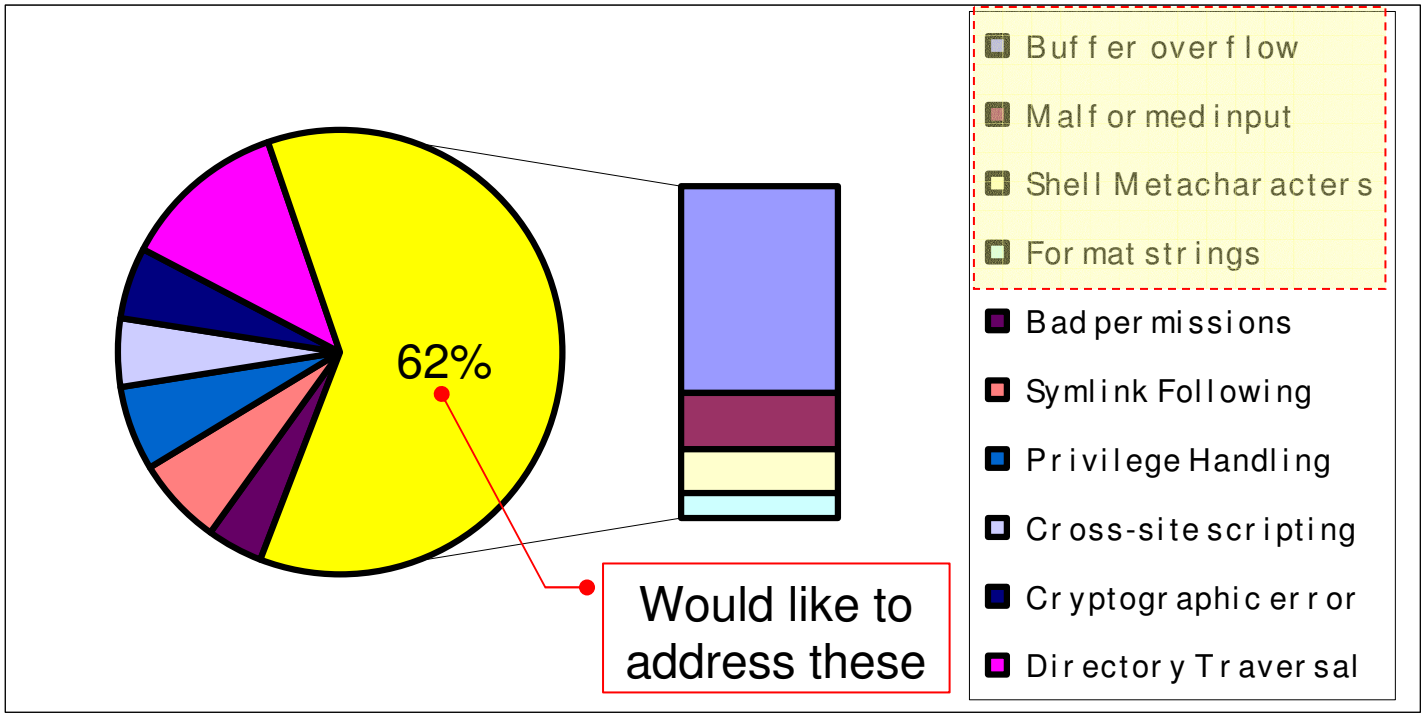
If failure to apply a patch costs 4 hours in System Admin Time to clean up the effects and patch the system, 2 hours in Developer Time to re-code any applications that have been affected by the patch or damage done by failure to patch and 30 minutes of downtime the cost of not patching is a whopping:

$820 + $410 + $500,000 = $**501,230**

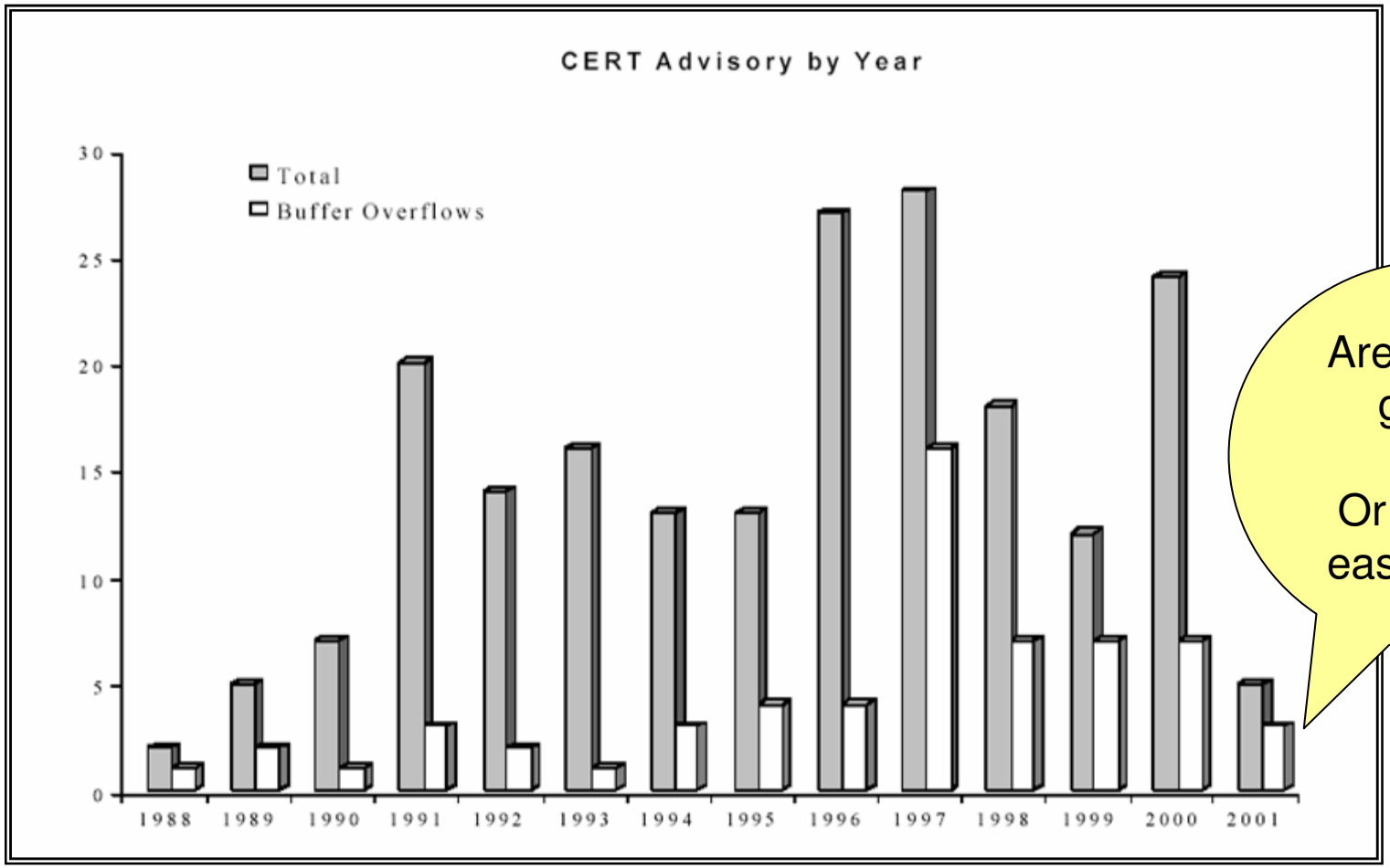
- Legal issues to consider
  - Who is responsible for lost and corrupt data? What to do with stolen credit card numbers, etc.?
  - Legislation demands compliance to security standards

Most Prevalent Classes

- SecurityFocus.com study of security reports in 2002
- Tried to identify most prevalent classes
- 3,582 CVE entries (1/2000 to 10/2002)
- Approximately 25% of the CVE was not classified



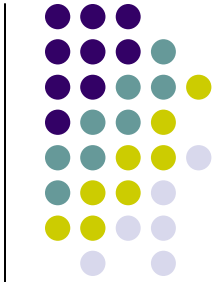
Security Vulnerabilities over Time



Are they **all** gone?

Or just the easy ones?

Focus of Our Work



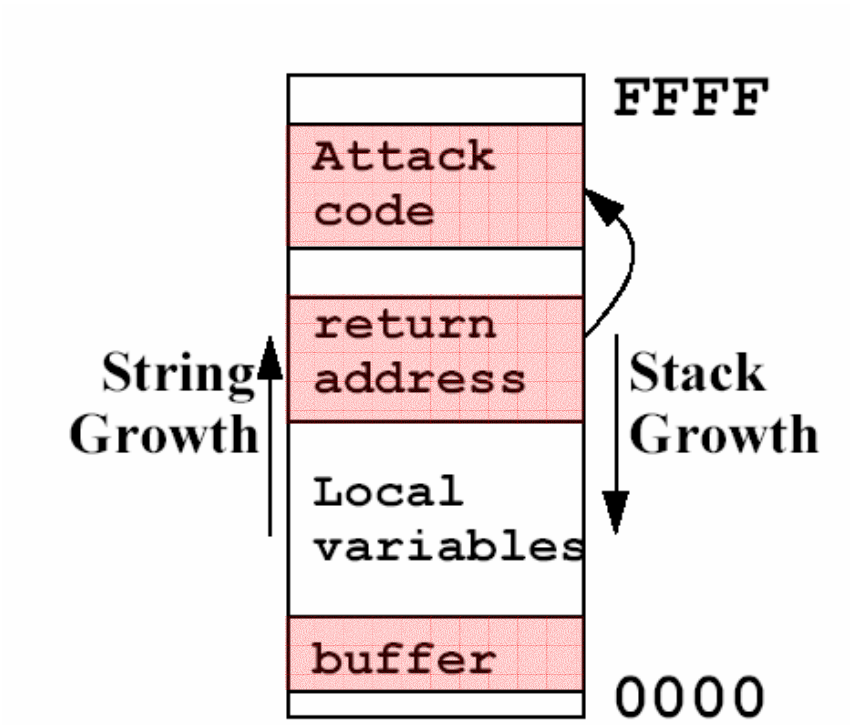
- We believe that tools are needed to detect security vulnerabilities
- We concentrate on the following types of vulnerabilities:
  - Buffer overruns
  - Format string violations
- Provide tools that are practical and precise

How Buffer Overruns Work

- Different flavors of overruns with different levels of complexity
- Simplest: overrun a static buffer
- There is no array bounds checking in C – hackers can exploit that
- Different flavors are descibed in detail in Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, [C.Cowan et al]
- We concentrate on overrunning *static buffers*

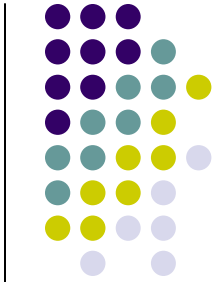
Don't want user data to be copied to static buffers!

Mechanics of a Simple Overrun



1. Arrange for suitable code to be available in program address space
   - usually by supplying a string with executable code
2. Get the program to jump to that code with suitable parameters loaded into registers & memory
   - usually by overwriting a return address to point to the string
3. Put something interesting into the exploit code
   - such as exec("sh"), etc.

How Format String Violations Work



- The "%n" format specifier – root of all evil
- Stores the number of bytes that are actually formatted:

```
printf("%.20x%n",buffer,&bytes_formatted);
```

- This is benign, but the following is not:

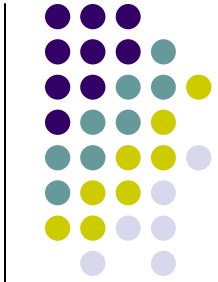
```
printf(argv[0]);
```

- Can use the power of "%n" to overwrite return address, etc.
- Requires some skill to abuse this feature
- In the best case – a crash, in the worst case – can gain control of the remote machine
- However the following is fine:

```
printf("%s", argv[0]);
```

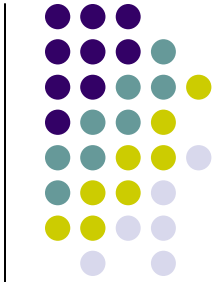
Don't want user data to be used as format strings!

Existing Auditing Tools



- Various specialized dynamic tools
  - Require a particular input/test case to run
  - Areas:
    - Network security
    - Runtime break-in detection
    - StackGuard for buffer overruns, many others
- Lexical scanners
  - Publicly available
    - RATS [securesoftware.com]
    - ITS4 [cigital.com]
    - pscan [open source] – simple format string violation finder
  - Typically imprecise:
    - Tend to inundate the user with warnings
    - Digging through the warnings is tedious
    - Discourages the user
- Can we do better with static analysis?

Talk Outline



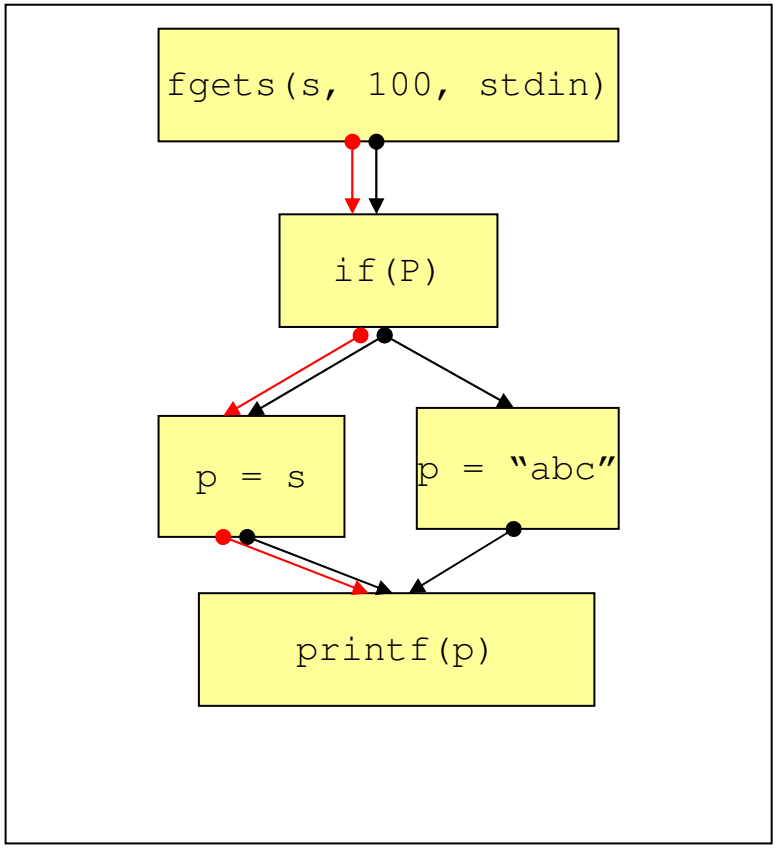
- Motivation: need better static analysis for security
- Detecting security vulnerabilities: existing approaches
- Static analysis: what are the components?
- Our approach: IPSSA + tools based on it
- Results and experience

Existing Static Approaches

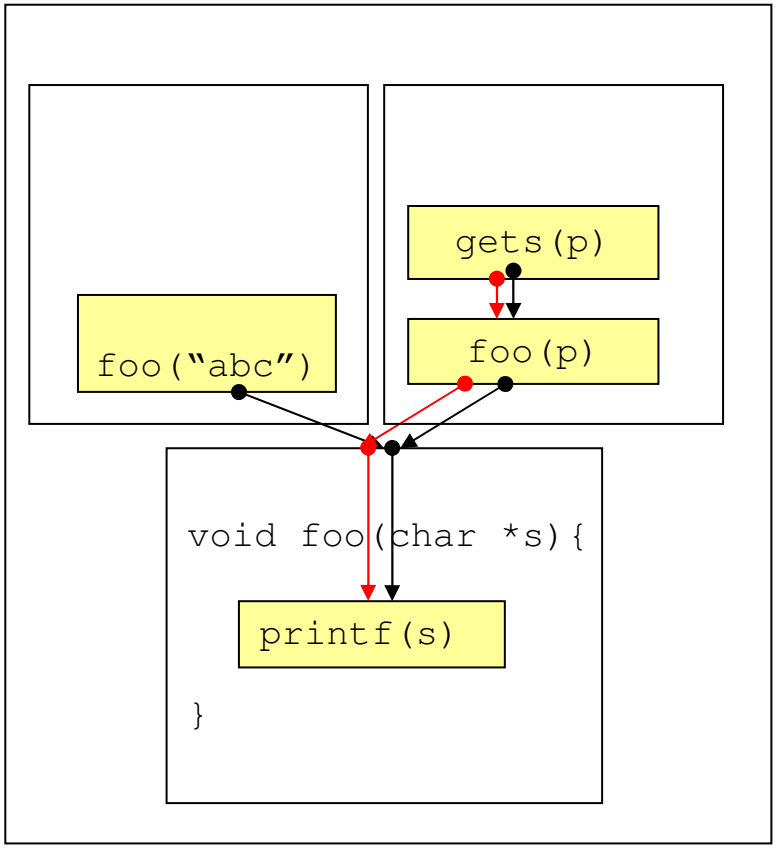
- A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities [D.Wagner]
  - Buffer overruns as an integer range analysis problem
  - Checked Sendmail 8.9.3: 4 bugs/44 warnings
  - Conclusion: following features are necessary to achieve better precision
    - Flow sensitivity
    - Pointer analysis
- Detecting Format String Vulnerabilities with Type Qualifiers [A.Aiken]
  - "Tainted" annotations, requires some, infers the rest
  - Conclusion: following features are necessary to achieve better precision
    - Context sensitivity
    - Field sensitivity

Flow-, Path- & Context Sensitivity

Flow and path sensitivity



Context sensitivity



Pointer Analysis: Major Obstacle

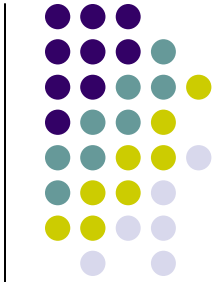
- Need it to represent data flow in C:

```
a = 2;
*p = 3;
...   ← is the value of a still 2?
```

- Yes if we can prove that `p` cannot point to `a`
- Should we put a flow edge from `3` to `a` to represent *potential* flow?
- Most existing pointer analysis approaches emphasize scalability and *not* precision
- Crucial realization:

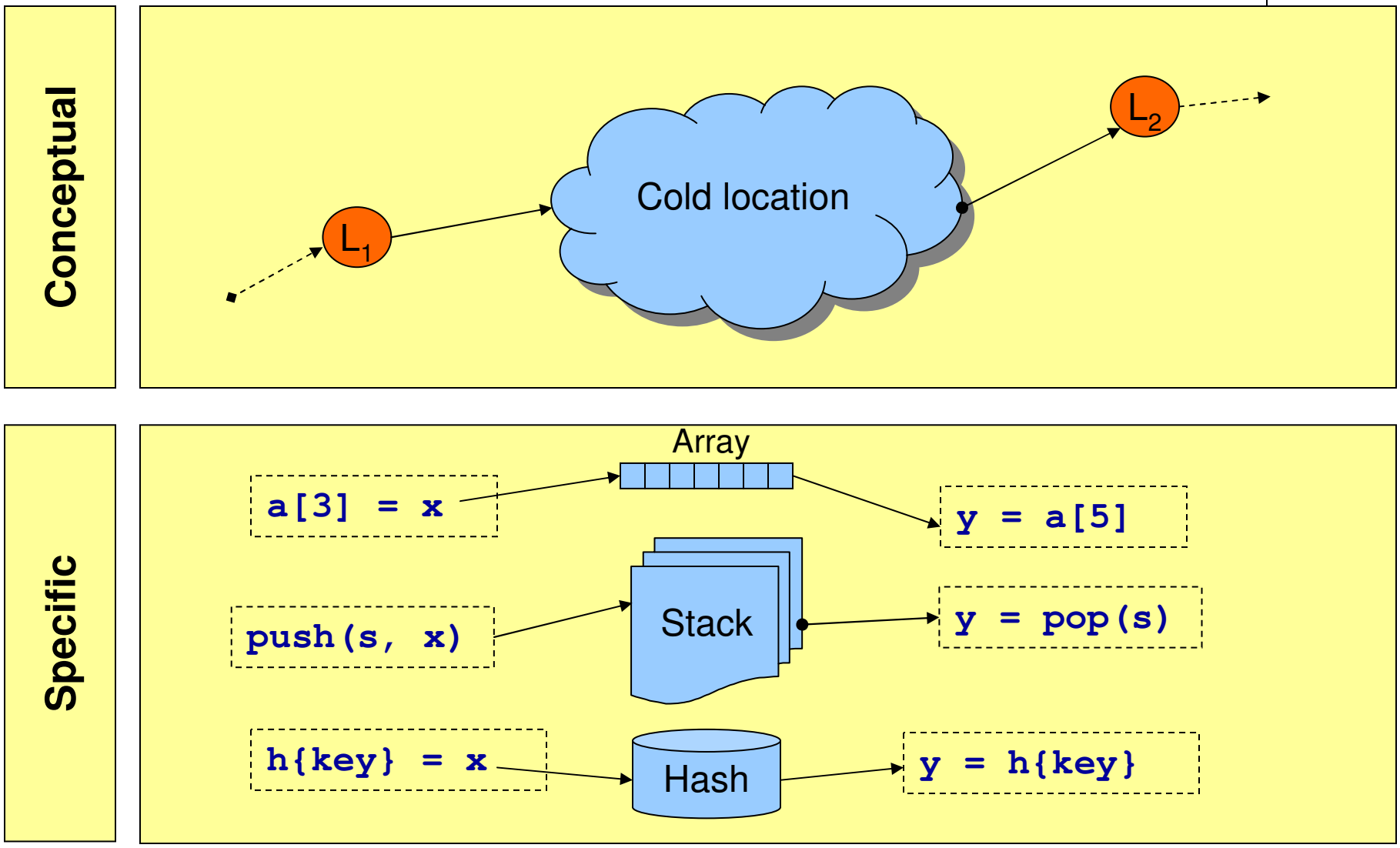
  We only need precision in certain places

To Achieve Precision…

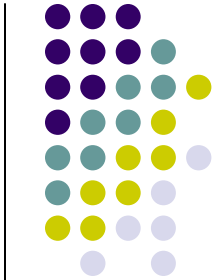


- Break the pointer analysis problem into two
- Precisely represent – "hot" locations
  - Local variables
  - Parameter passing
  - Field accesses and dereferences of parameters and locals
- All the rest if "cold"
  - Data structures
  - Arrays
  - etc.

Hot vs Cold Locations



Putting it All Together: Precision Requirements



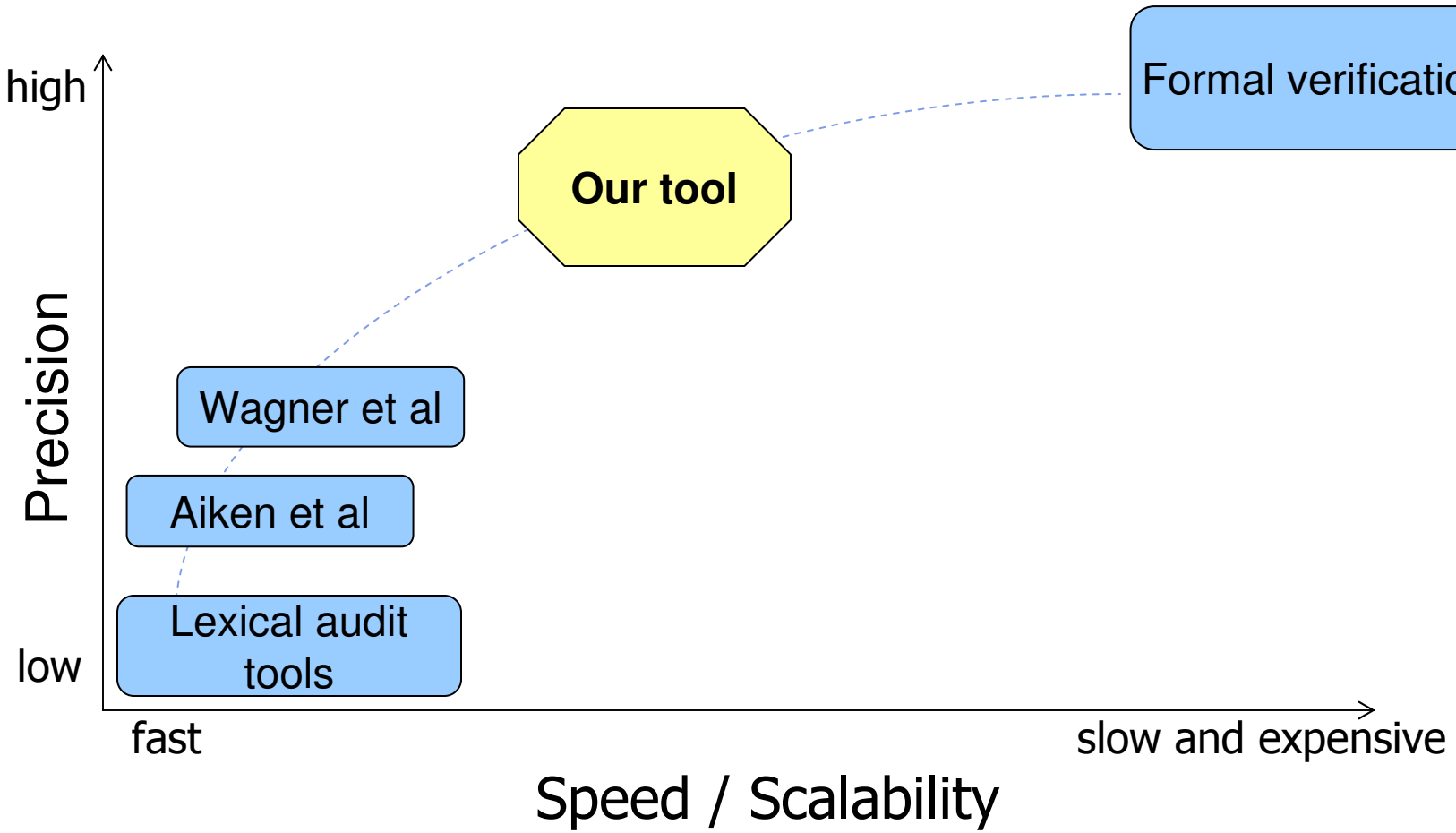
Wagner et al.	Aiken et al.
•Flow sensitivity •Pointer analysis	•Field sensitivity •Context sensitivity

+

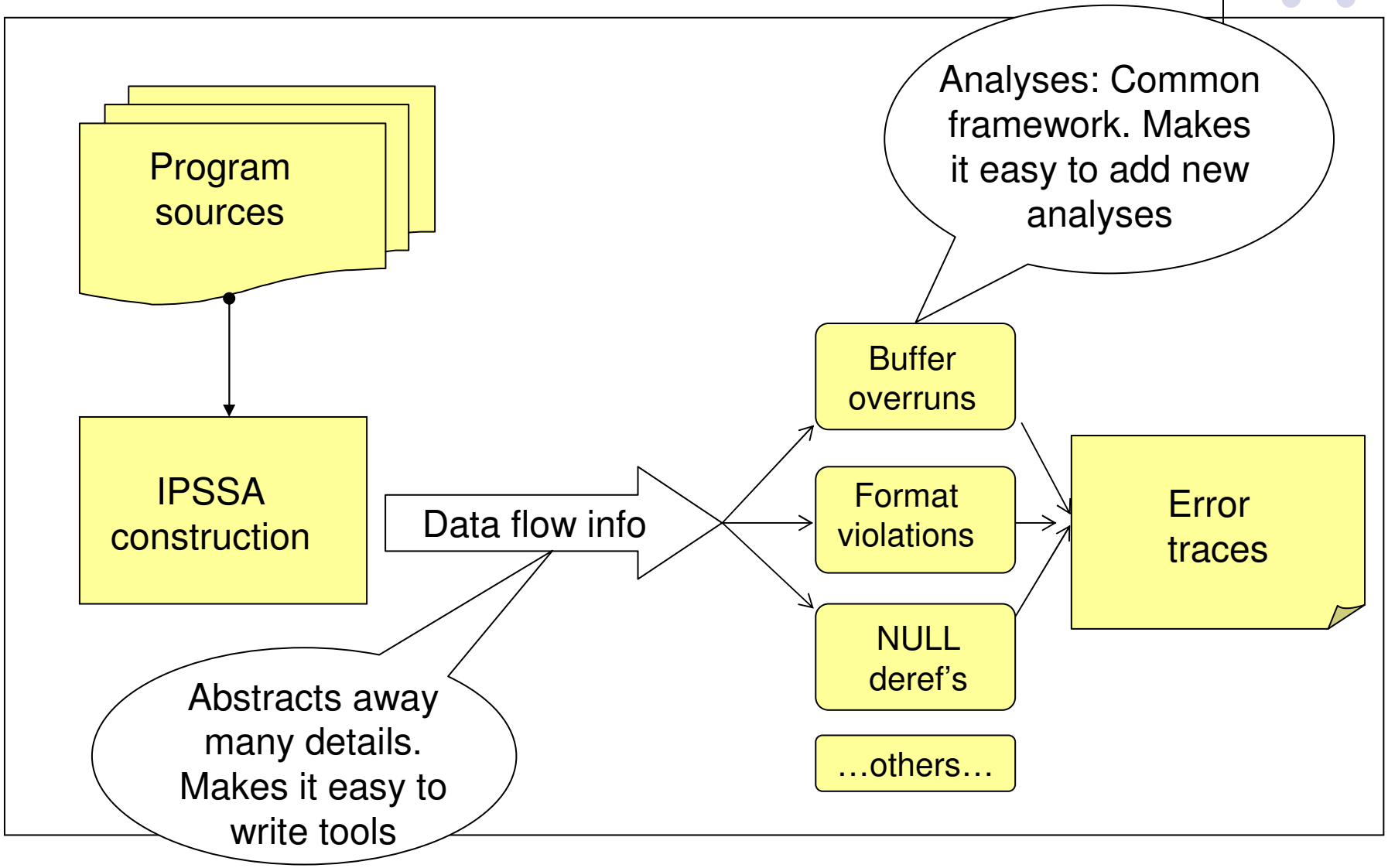
And also…

- Ability to analyze code scattered among many functions and files efficiently
  - This is where hard bugs hide
- Path-sensitivity
- Precise representation of library routines (Wagner, Aiken) such as
  - `strcpy`, `strncpy`, `strtok`, `memcopy`, `sprintf`, `snprintf`
  - `fprintf`, `printf`, `fgets`, `gets`
- Support features of C
  - Pass-by-reference semantics
  - `varargs` and `va_list` treatment
  - Function pointers

Tradeoff: Scalability vs Precision



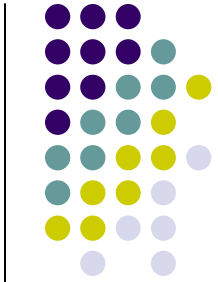
Our Framework



To Summarize:
New Program Representation: IPSSA

- Intraprocedurally
  - SSA – static single assignment form
  - Local pointer resolution: pointers are resolved to scalars, new names are introduced
- Interprocedurally
  - Parameter mapping
  - Globals treated as parameters
  - Side effects of calls are represented explicitly
- Hot vs Cold locations
  - Hot locations are represented precisely
  - Cold locations are multiple locations "lumped" together
- Models for system functions

Models of System Functions

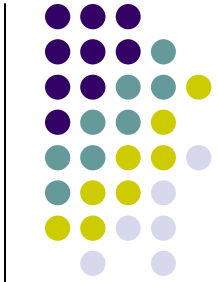


- Excerpt from a model specification file

```
tainted io char* gets(non_null char[] s){
     s[] = taint;
     return (s, NULL);
};

tainted io char* getenv(non_null char[] s){
     ret_loc^ = taint;
     return (unknown, NULL);
};

char* sprintf(char[] buf, non_tainted const char[] format, void* ...){
     buf^ = ...^;
     return buf;
};

char* snprintf(char[] buf, int sz, non_tainted const char[] format, void* ...){
     buf^ =  ...^;
     return buf;
};

io void fprintf(non_null FILE* file, non_tainted char[] format, void* ...){
     safe(...^);
};
```

- `non_tainted` qualifiers, explicit `taint` variable
- `varargs` are represented by "..."
- Pass-by-reference representation

Analysis Based on IPSSA



1. Start at sources of user input (*roots*) such as
   - `argv[]` elements
   - sources of input: `fgets`, `gets`, `recv`, `getenv`, etc.
2. Follow data flow provided by IPSSA until a *sink* is found
   - Buffer of statically defined length
   - Vulnerable procedures: `printf`, `fprintf`, `snprintf`, `vsnprintf`
3. Test path feasibility using predicates (optional step)
4. Report bug, record path

Example: Tainting Violation in muh

muh.c:839

```
0838            s = ( char * )malloc( 1024 );
0839            while( fgets( s, 1023, messagelog ) ) {
0840                if( s[ strlen( s ) - 1 ] == '\n' ) s[ strlen( s )...
0841                irc_notice( &c_client, status.nickname, s );
0842            }
0843            FREESTRING( s );
0844
0845            irc_notice( &c_client, status.nickname, CLNT_MSGLOGEND );
```

irc.c:263

```
257 void irc_notice(connection_type *connection, char nickname[], char *format,
 ... )
258 {
259     va_list va;
260     char buffer[ BUFFERSIZE ];
261
262     va_start( va, format );
263     vsnprintf( buffer, BUFFERSIZE - 10, format, va );
264     va_end( va );
```

Example: Buffer Overrun in `gzip`

gzip.c:593

```
0589      if (to_stdout && !test && !list && (!decompress || ...
0590          SET_BINARY_MODE(fileno(stdout));
0591      }
0592          while (optind < argc) {
0593          treat_file(argv[optind++]);
```

gzip.c:716

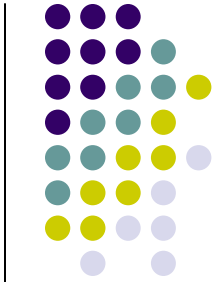
```
0704 local void treat_file(iname)
0705      char *iname;
0706 {
                                  ...
0716      if (get_istat(iname, &istat) != OK) return;
```

gzip.c:1009

```
0997 local int get_istat(iname, sbuf)
0998      char *iname;
0999      struct stat *sbuf;
1000 {
                                  ...
1009      strcpy(ifname, iname);
```

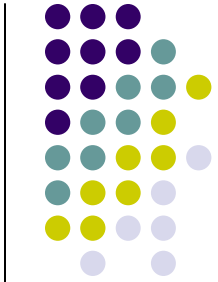
Need to have a model of `strcpy`

Recurring Patterns: Lessons Learned



- "Hard" violations pass through many procedures
  - About 4 on average
  - Not surprising – the further away a root is from a sink, the harded it is to find manually
- "Harder" violations pass through many files
- Relatively few unique root-sink *pairs*
- But… potentially many more root-sink *paths*

Do We Need Predicates?



- Predicates are *sometimes* important in reducing false positive ratio
- Hugely depends on the application: help with NULLs
- A few places where they matter in the security analysis

util.c (lhttpd 0.1)

```
109     while(!feof(in))
110     {
111         getfileline(tempstring, in);
112
113         if(feof(in)) break;
114         ptr1 = strtok(tempstring, "\" \t");
```

```
160     while(!feof(in))
161     {
162         getfileline(tempstring, in);
163
164         if(feof(in)) break;
165         ptr1 = strtok(tempstring, "\"\t ");
166         ptr2 = strtok(NULL, "\"\t ");
```

- Predicates are sometimes needed in function models for precision
- When called with NULL as the first argument, strtok returns portions of the string previously passed into it
- Otherwise, the passed in string is stored internally

- *No* flow between `tempstring` on line 114 and 165
- There *is* flow between tempstring and `ptr2` on lines 165 and 166

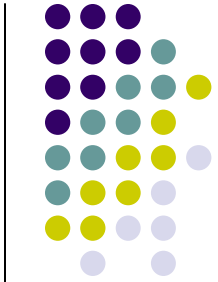
Summary of Experimental Results

Program	Version #	LOC	Procedures
lhttp	0.1	888	21
bftpd	1.0.11	2,946	47
trollftpd	1.26	3,584	48
man	1.5h1	4,139	83
cfingerd	1.4.3	5,094	66
muh	2.05d	5,695	95
gzip	1.2.4	8,162	93

- 7 server-type programs
- Contained many violations previously reported on SecurityFocus and other security sites

Program name	Total number of warnings	Buffer overruns	Format string violations	False positives	Number of sources	Number of sinks	Definitions spanned	Procedures spanned	Tool's runtime sec
lhttpd	1	1		20 (w/o pr	4	1	7	4	7.08
bftpd	2	1	1		5	2	5,7	1,3	2.34
trollftpd	1	1			4	1	23	5	8.52
man	1	1			3	1	6	4	9.67
cfingerd	1		1		4	1	10	4	7.44
muh	1		1		3	1	7	3	7.52
gzip	1	1			3	1	7	5	2.03

Conclusions



- Outlined the need for static pointer analysis to detect security violations
- Presented a program representation designed for bug detection
- Described how it can be used in an analysis to find security violations
- Presented experimental data that demonstrate the effectiveness of our approach
- More details: there is a paper available:

`http://suif.stanford.edu/~livshits/papers/fse03.ps`

Thanks for listening!