Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs

by

V.Benjamin Livshits and Monica S. Lam
{livshits, lam}@cs.stanford.edu
SUIF Group
CSL, Stanford University

Background

- Software systems are getting bigger
  - Harder to develop
  - Harder to modify
  - **Harder to debug and test**
- Bug detection needs to be automated
- Classes of automatic error detection tools
  - Memory consistency errors
  - Locking errors
  - Resource consistency: files, sockets, etc.
  - Application-specific logical properties and constraints
  - `NULL` pointer dereferences
  - **Potential security violations**
  - etc.

Motivating Examples

- Bugs from the security world:
  - Two previously known security vulnerabilities
    - Buffer overrun in **`gzip`**, compression utility
    - Format string violation in **`muh`**, network game
- Unsafe use of user-supplied data
  - **`gzip`** copies it to a statically-sized buffer, which may result in an overrun
  - **`muh`** uses it as the format argument of a call to **`vsnprintf`** – user can maliciously embed **`%n`** into format string

Buffer Overrun in `gzip`

gzip.c:593

```
0592      while (optind < argc) {
0593          treat_file(argv[optind++]);
0594      }
```

gzip.c:716

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

gzip.c:1009

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

Need a model of `strcpy`

gzip.c:233

```
0233 char ifname[MAX_PATH_LEN]; /*input file name*/
```

Format String Violation in `muh`

muh.c:839

```
0838              s = ( char * )malloc( 1024 );
0839              while( fgets( s, 1023, messagelog ) ) {
0841                 irc_notice(&c_client, status.nickname, s);
0842              }
0843              FREESTRING( s );
```

irc.c:263

```
257 void irc_notice(con_type *con, char nick[],
                                  char *format, ... ){
259     va_list va;
260     char buffer[ BUFFERSIZE ];
261
262     va_start( va, format );
263     vsnprintf( buffer, BUFFERSIZE - 10, format, va );
```

Looking at Applications…

- **Some** security bugs are easy to find
  - There is a number of lexical source auditing tools
  - We are *not* after the easy bugs
- Programs have security violations despite code reviews and years of use
- Common observation about hard errors:
  - Errors on interface boundaries – need to follow data flow between procedures
  - Errors occur along complicated control-flow paths: need to follow long definition-use chains

Need to Understand Data Flow

- Both security examples involve complex flow of data
- Main problem: To track data flow in C/C++ need to understand relationships between pointers
- Basic example:

```
*p = 2
```

- Indirect stores can create new data assignments
- Conservatively would need to assign **2** to everything
- Pointer analysis to determine what may be affected

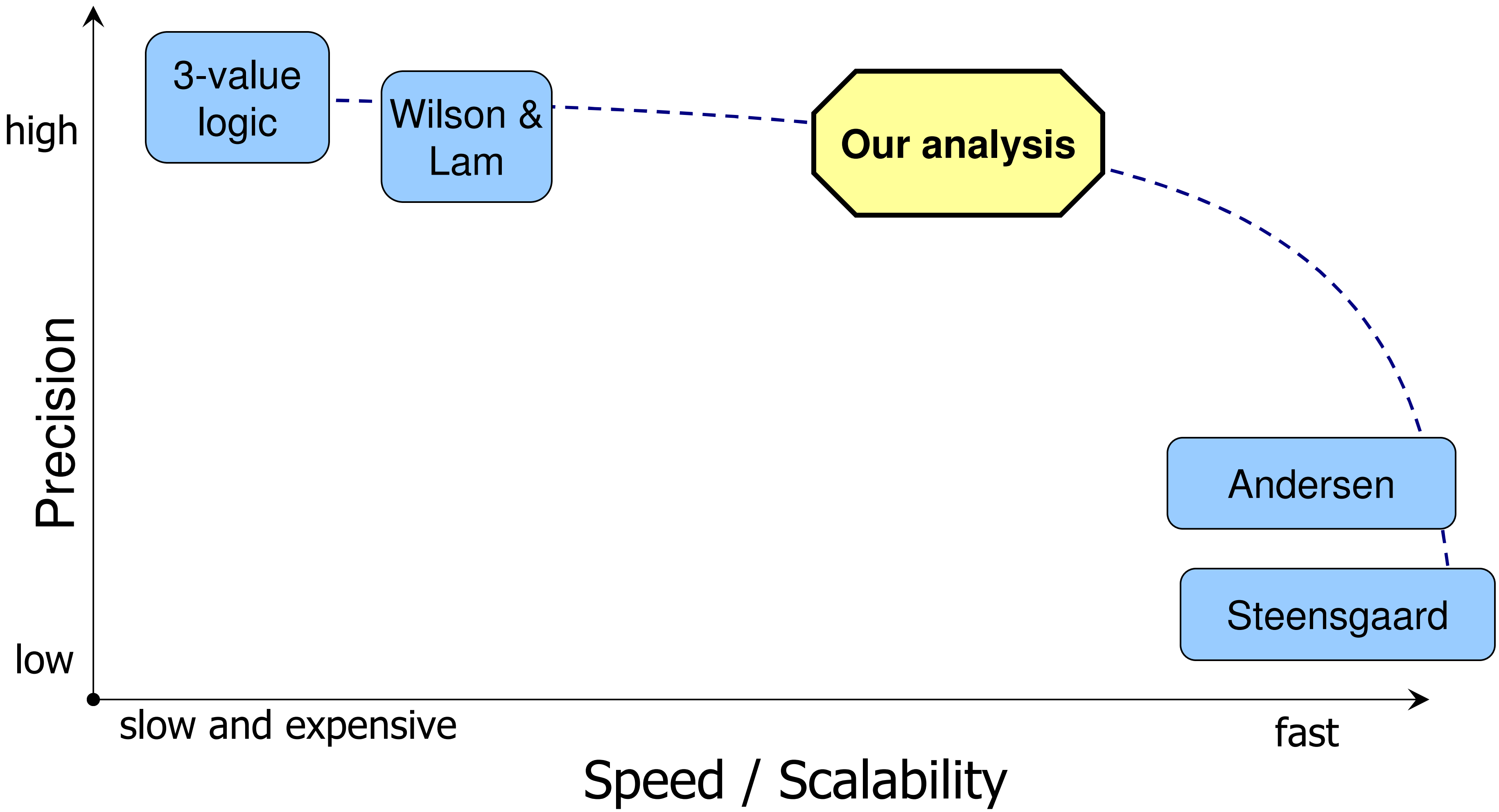
Fast Pointer Analyses

- Typical sound pointer analyses: emphasize scalability over precision
- Steensgaard's [1996]
  - Flow- and context insensitive
  - Essentially linear time
  - Used to analyze Microsoft Word – 2.2 MLOC
- Andersen's [1994] and CLA [2001]
  - More precise than Steensgaard's
  - CLA – optimized version of Andersen's with fields – 1 MLOC a second
  - Still flow- and context-insensitive
- Others…

More Precise Analyses?

- Flow- and context-insensitive approaches are fast
- But generally too imprecise for error detection tools:
  - Flow- and context-insensitive – all possible flows through a procedure and all calling contexts are merged together
  - Lack of flow- and context-sensitivity can result in a very high number of false positives
- Flow- and context-sensitive techniques are not known to scale
  - Sagiv et.al., *Parametric shape analysis via 3-valued logic, 1999,* everything-sensitive
  - Wilson & Lam, *Efficient context-sensitive pointer analysis for C programs,* 1995, flow- and context-sensitive

Tradeoff: Scalability vs Precision



Our Approach to Pointers

- Propose a **hybrid** approach to pointers – maintain precision selectively

- Analyze *very* precisely:
  - Local variables
  - Procedure parameters
  - Global variables
  - …their dereferences and fields
- These are essentially *access paths*, i.e. $p.next.data$.

- Break all the rest into coarse equivalence classes
- Represent the rest by *abstract locations*:
  - Recursive data structures
  - Arrays
  - Locations accessed through pointer arithmetic
  - etc.

Two Levels of Pointer Analysis

- Regular assignments result in strong updates

`x = 1;`	$x_0 = 1$
`x = 2;`	$x_1 = 2$
`y = x;`	$y_0 = x_1$

x is **2**

- Break all locations into equivalence classes – ECRs [Steensgaard, 1996]
- Abstract memory locations correspond to ECRs
- Assignments to abstract memory locations – weak updates
- Conservative approach – don't overwrite old data

`A[i] = 1;`	$m_0 = 1$
`A[j] = 2;`	$m_1 = \phi(m_0, 2)$
`b = A[k];`	$b_0 = m_1$

Either **1** or **2**

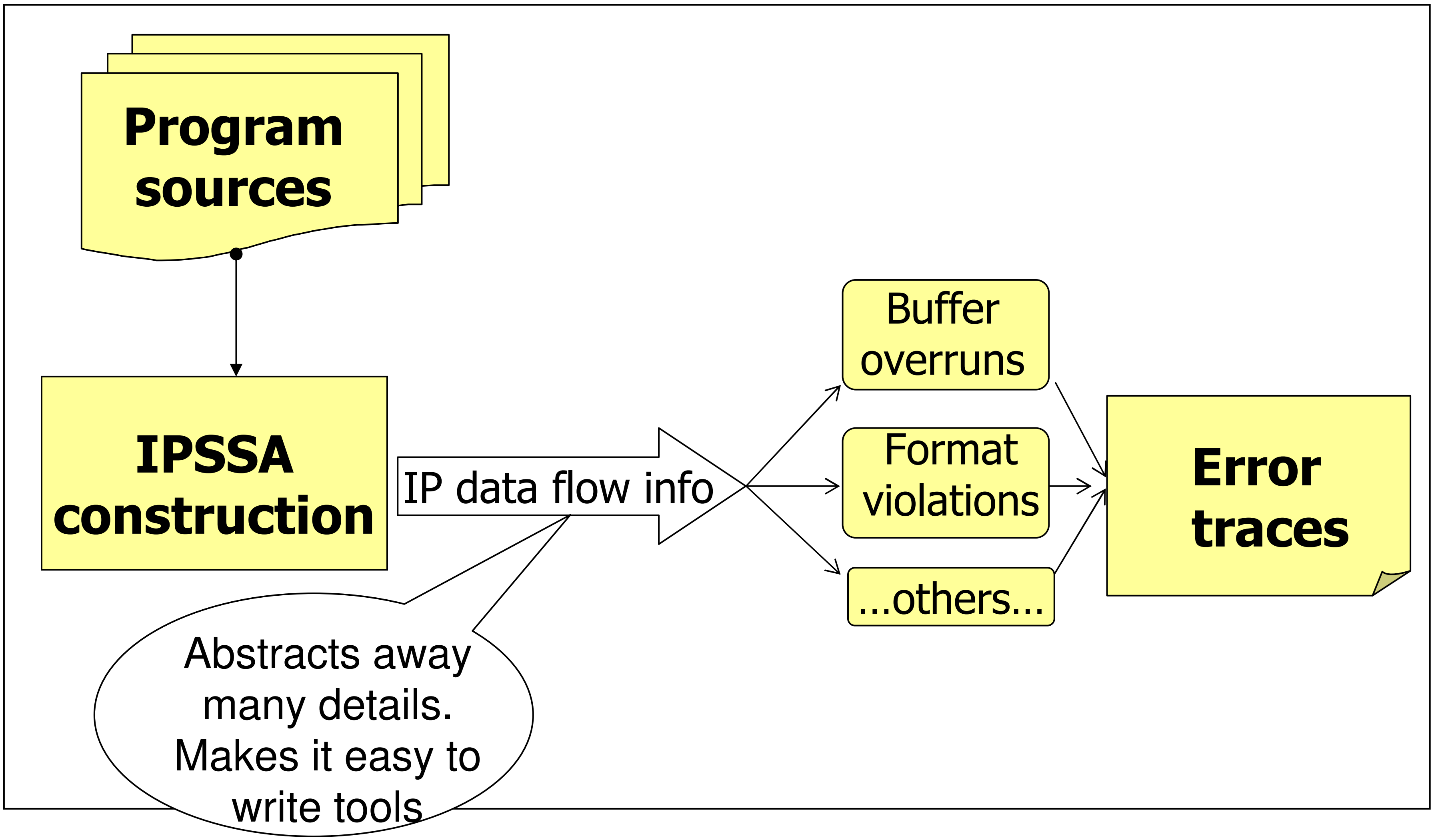
Error Detection Tools

- Existing tools need to infer data flow:
  - Intrinsa
  - Dawson
  - Others
- Lack of precision – more false warnings
- Too many false warnings – don't get used
- Lack of soundness guarantee

Talk Outline

- Motivation: pointer analysis for error detection
- Pointer analysis and design of IPSSA – InterProcedural SSA, associated algorithms
- Using data flow information provided by IPSSA for security applications
- Results and experience: study of security vulnerability detection tool

Our Framework

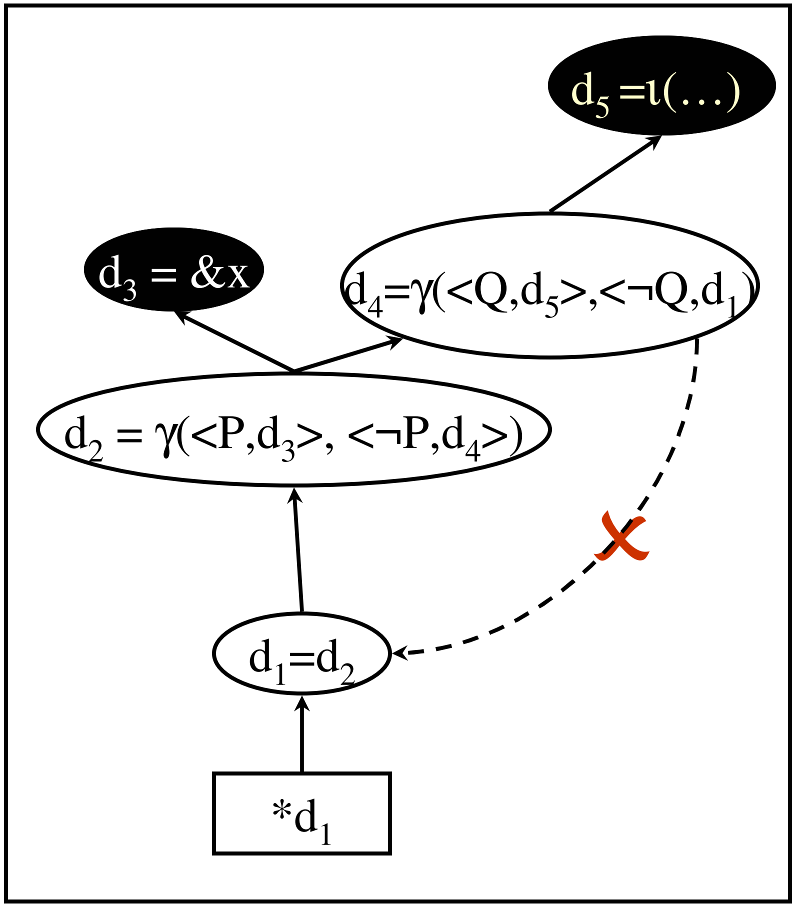


IPSSA – Intraprocedurally

- Intraprocedurally: an extension of Gated SSA
- Gated SSA [Tu, Padua 1995]
  - Give new names (subscripts) to definitions – solves flow-sensitivity problem
  - Create predicated γ functions – combine reaching definitions of the same variable
- Important extension provided by IPSSA:
  - Our version of pointer analysis – *pointer resolution*
  - Replace indirect pointer dereferences with direct accesses of potentially new temporary locations

Pointer Resolution Algorithm

- Iterative process
- At each step definition *d* is being dereferenced:
  - T*erminal resolution node* – resolve and stop
  - Otherwise follow all definitions on RHS
- *Occurs-check* to deal with recursion
- See paper for complete rewrite rules



Example of Pointer Resolution

```
int a=0,b=1;
int c=2,d=3;

if(Q){
    p = &a;
}else{
    p = &b;
}

c  = *p;

*p = d;
```

Load resolution (c = *p;)

Store resolution (*p = d;)

$a_0 = 0,\ b_0 = 1$

$c_0 = 2,\ d_0 = 3$

$p_1 = \&a$

$p_2 = \&b$

$p_3 = \gamma(<Q,\ p_1>,\ <\neg Q,\ p_2>)$

$c_1 = \gamma(<Q,\ a_0>,\ <\neg Q,\ b_0>)$

$a_1 = \gamma(<Q,\ d_0>,\ <\neg Q,\ a_0>)$

$b_1 = \gamma(<Q,\ b_0>,\ <\neg Q,\ d_0>)$

Pointer Resolution Rules

- When resolving definition d, next step depends on RHS of d
- Expressed as conditional rewrite rules
- A few sample rules:
  - $d = \&x$, result is x
  - $d = \iota(\ldots)$, result is d^
  - $d = \gamma(<P_1, d_1>,\ldots,<P_n, d_n>)$, follow $d_1 \ldots d_n$
- Refer to the paper for details

Interprocedural Algorithm

- Consider program in a bottom-up fashion, one strongly-connected component (SCC) of the call graph at a time
- **Unsound** unaliasing assumption – assume that we can't reach the same location through two different parameters
- For each SCC, within each procedure:
  1. Resolve all pointer operations (loads and stores)
  2. Create links between formal and actual parameters
  3. Reflect stores and assignments to globals at call sites
- Iterate within SCC until the representation stabilizes

Unsound Unaliasing Assumption

	A1: No aliased parameters	A2: No aliased abstract locations
Assumption	Locations accessible through different parameters are distinct	Things pulled out of an abstract location is not aliased
Justification	Matches how good interfaces are written	Holds in most usage cases
Consequence	Context-independent procedure summaries	Give unique names when we get data from abstract location

Interprocedural Example

- Data flow in and out of functions:
  - Create links between formal and actual parameters
  - Reflect stores and assignments to globals at the callee
- Can be a lot of work – many parameters and side effects

```
int f(int* p){
        *p = 100;
}

int main(){
        int x = 0;
        int *q = &x;

   c:   f(q);
}
```

$p_0 = \iota(\langle c, q_0 \rangle)$

$p\hat{}_1 = 100$

$x_0 = 0$

$q_0 = \&x$

$x_1 = \rho(\langle f, 100 \rangle)$

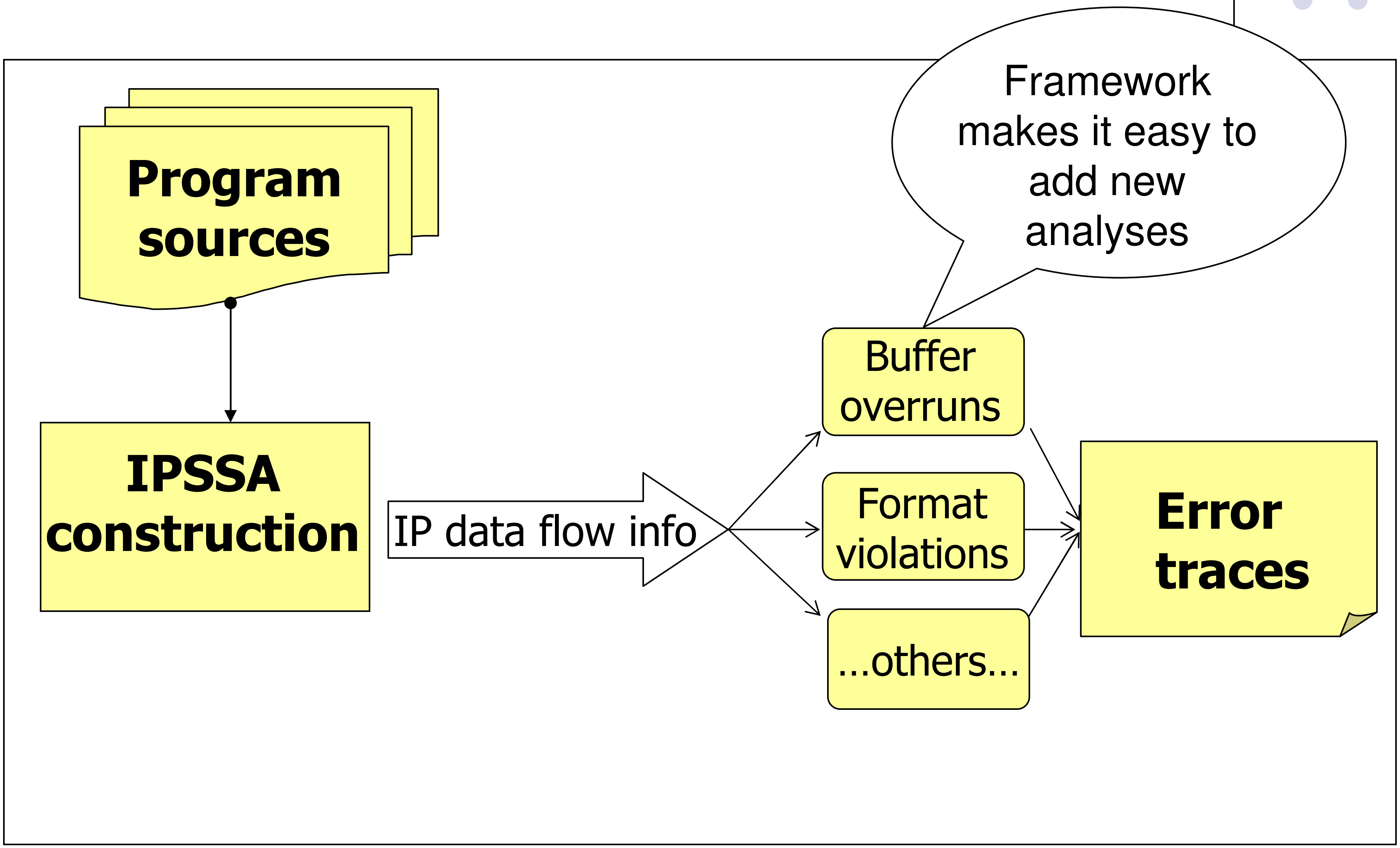
Formal-actual connection for call site **c**

Reflect store inside of **f** within **main**

Summary of IPSSA Features

- Intraprocedural
  - Pointers are resolved, replaced w/direct accesses
  - Hybrid pointer approach: two levels of pointers
  - Assignments to abstract memory locations result in weak updates
  - Treat structure fields as separate variables
- Interprocedural
  - Process program bottom up, one SCC at a time
  - Unsound unaliasing assumption to speed up the analysis

Our Framework



Our Application: Security

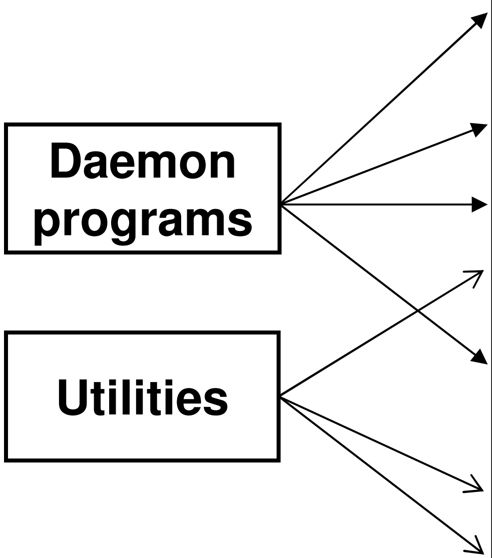
- Want to detect
  - A class of buffer overruns resulting from copying user-provided data to statically declared buffers
  - Format string violations resulting from using user-provided data as the format parameter of **`printf, sprintf, vsnprint`**, etc.
  - Note: *not* detecting overruns produced by accessing string buffers through indices, that would require analyzing integer subscripts
- Want to report
  - Detailed error path traces, just like with **`gzip`** and **`muh`**
  - (Optional) Reachability predicate for each trace

Analysis Formulation

1. Start at *roots* – sources of user input such as
   - **`argv[]`** elements
   - Input functions: **`fgets`**, **`gets`**, **`recv`**, **`getenv`**, etc.
2. Follow data flow chains provided by IPSSA: for every definition, IPSSA provides a list of its uses
   - Achieve path-sensitivity as a result
   - Match call and return sites – context-sensitivity
3. A *sink* is a potentially dangerous usage such as
   - A buffer of a statically defined length
   - A format argument of vulnerable functions: **`printf`**, **`fprintf`**, **`snprintf`**, **`vsnprintf`**
4. Report bug, record full path

Experimental Setup

- Implementation
  - Uses SUIF2 compiler framework
  - Runtime numbers are for Pentium IV 2GHz machine with 2GB of RAM running Linux



Program	Version	LOC	Procedures	IPSSA constr. time, seconds
lhttpd	0.1	888	21	5.2
polymorph	0.4.0	1,015	19	1.0
bftpd	1.0.11	2,946	47	3.2
trollftpd	1.26	3,584	48	11.3
man	1.5h1	4,139	83	29.3
pgp4pine	1.76	4,804	69	17.5
cfingerd	1.4.3	5,094	66	15.5
muh	2.05d	5,695	95	20.4
gzip	1.2.4	8,162	93	17.0
pcre	3.9	13,037	47	22.4

Summary of Experimental Results

Program name	Total # of warnings	Buffer over-runs	Format string vulner.	False positives	Defs spanned	Procs spanned	Tool's runtime sec
lhttpd	1	1	0	0	24	14	99
polymorph	2	2	0	0	7,8	3	2.4
bftpd	2	[illegible]	[illegible]	0	5, 7	1, 3	2.3 s
trollftpd	1	[illegible]	[illegible]	0	23	5	8.5 s
man	1	1	0	0	6	4	9.6 s
pgp4pine	4	4	0	[illegible]	5, 5	3, 3, 3, 3	27.1 s
cfingerd	1	0	1	[illegible]	10	4	7.4 s
muh	1	0	1	0	7	3	7.5 s
gzip	1	1	0	0	7	5	2.0 s
pcre	1	0	0	1	6	4	9.2 s
Total	**15**	**11**	**3**	**1**	Previously unknown: **6**		

Many definitions

Many procedures

False Positive in `pcre`

- Copying "tainted" user data to a statically-sized buffer may be unsafe
- Turns out to be safe in this case

Tainted data

```
sprintf(buffer, "%.512s", filename)
```

Limits the length of copied data. Buffer is big enough!

Conclusions

- Outlined the need for static pointer analysis for error detection
- IPSSA, a program representation designed for bug detection and algorithms for its construction
- Described how analysis can use IPSSA to find a class of security violations
- Presented experimental data that demonstrate the effectiveness of our approach