

Static Analysis to Improve Software Quality

Jeff Foster Mike Hicks
University of Maryland

Software Quality Today

Trustworthy Computing is computing that is available, reliable, and secure as electricity, water services and telephony....No Trustworthy Computing platform exists today.

-- Bill Gates, January 15, 2002
(highest priority for Microsoft)

[T]he national annual costs of an inadequate infrastructure for software testing is estimated to range from \$22.2 to \$59.5 billion.

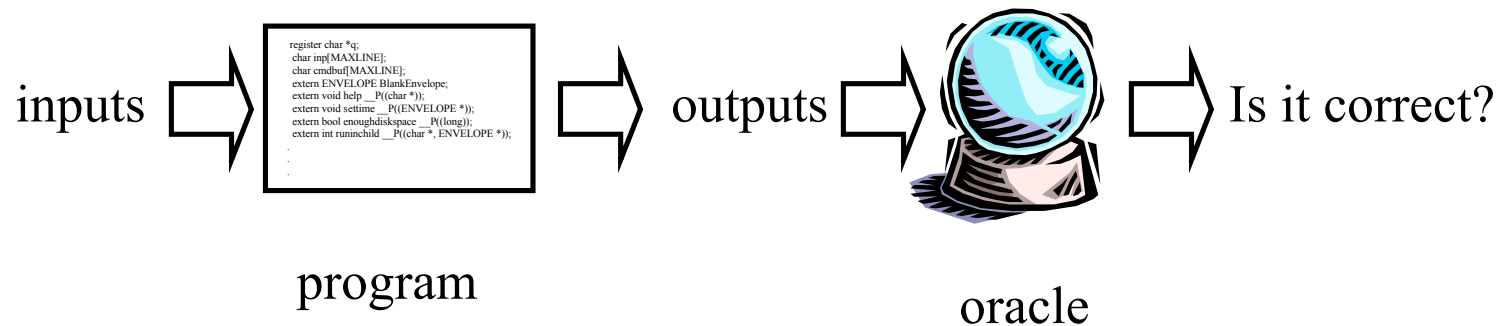
-- NIST Planning Report 02-3, May 2002

Conclusions?

- Software is buggy
 - It's hard to ensure that it's reliable
 - ...and doing so is important

Current Practice

- Testing
 - Make sure program runs correctly on set of inputs



- Drawbacks: Expensive, difficult, hard to cover all code paths, no guarantees

Current Practice (cont'd)

- Code Auditing
 - Convince someone else your source code is correct
 - Drawbacks: Expensive, hard, no guarantees



```
register char *q;
char inp[MAXLINE];
char cndbuf[MAXLINE];
extern ENVELOPE BlankEnvelope;
extern void help __P(char *);
extern void settime __P(ENVELOPE *);
extern bool enoughdiskpace __P(long);
extern int runinchild __P(char *, ENVELOPE *);
extern void checksmtpattack __P(volatile int *, int, char *, ENVELOPE *);

if (fileno(OutChannel) != fileno(stdout))
{
    /* arrange for debugging output to go to remote host */
    (void) dup2(fileno(OutChannel), fileno(stdout));
}
settime(c);
peerhostname = RealHostName;
if (peerhostname == NULL)
peerhostname = "localhost";
CurHostName = peerhostname;
CurSmtpClient = macvalue('_', c);
if (CurSmtpClient == NULL)
CurSmtpClient = CurHostName;

setprctitle("server %s startup", CurSmtpClient);
#if DAEMON
if (LogLevel > 1)
{
    /* log connection information */
    sm_syslog(LOG_INFO, NOOD,
        "SMTP connect from %100s (%100s)",
        CurSmtpClient, anynet_ntoa(&RealHostAddr));
}
#endif

/* output the first line, inserting "ESMTP" as second word */
expandSmtpGreeting, inp, sizeof inp, c);
p = strchr(inp, '\n');
if (p != NULL)
*pp++ = '\0';
id = strchr(inp, ':');
if (id == NULL)
id = &inp[strlen(inp)];
cmd = p == NULL ? "220 %s ESMTP%" : "220-%s ESMTP%" ;
message(cmd, id - inp, inp, id);

/* output remaining lines */
while ((id = p) != NULL && (p = strchr(id, '\n')) != NULL)
{
    *pp++ = '\0';
    if (isascii(*id) && isspace(*id))
```

```
cmd <- &cndbuf[sizeof cndbuf - 2])
*cmd++ = *p++;
*cmd = '\0';

/* throw away leading whitespace */
while (isascii(*p) && isspace(*p))
p++;

/* decode command */
for (c = CmdTab, c-<cmdname != NULL; c++)
{
    if (!strcmp(c->cmdname, cndbuf))
        break;
}

/* reset errors */
errno = 0;

/* Process command.
** If we are running as a null server, return 550
** to everything.
*/

if (nullserver)
{
    switch (c->cmdcode)
    {
        case CMDQUIT:
        case CMDHELO:
        case CMDEHLO:
        case CMDNOOP:
            /* process normally */
            break;
        default:
            if (++badcommands > MAXBADCOMMANDS)
                sleep(1);
            return("550 Access denied");
            continue;
    }
}

/* non-null server */
switch (c->cmdcode)
{
    case CMDMAIL:
    case CMDEXPN:
    case CMDVRFY:
```

```
while (isascii(*p) && isspace(*p))
p++;
if (*p == '\0')
break;
kp = p;

/* skip to the value portion */
while ((isascii(*p) && isalnum(*p)) || *p == '-')
p++;
if (*p == '\n')
{
    *pp++ = '\0';
    vp = p;
}

/* skip to the end of the value */
while (*p != '\0' && *p != '-' &&
(isascii(*p) && iscntrl(*p)) &&
*p != '\n')
p++;
}

if (*p != '\0')
*pp++ = '\0';

if (Tid19, 1)
print("RCPT: got arg %s", "%s", n, kp,
vp == NULL ? "null" : vp);

rept_esmtp_args(a, kp, vp, c);
if (Errors > 0)
break;
}
if (Errors > 0)
break;

/* save in recipient list after ESMTP mods */
a = recipient(a, &c->e_sendqueue, 0, c);
if (Errors > 0)
break;

/* no errors during parsing, but might be a duplicate */
c->e_to = a->q_paddr;
if (bitset(QBADADDR, a->q_flags))
{
    message("250 Recipient ok");
    bitset(QQUEUEUP, a->q_flags);
    /* (will queue) - */;
}
nrpts++;
}
else
{
    /* punt -- should keep message in ADDRESS... */
}
```

And If You're Worried about Security...

A **malicious adversary** is trying to exploit anything you miss!



What more can we do?

Tools for Software Quality

- Build tools that analyze source code (static analysis)
 - Reason about all possible runs of the program
- Check limited but very useful properties
 - Eliminate categories of errors
 - Let people concentrate on the deep reasoning
- Develop programming models
 - Avoid mistakes in the first place
 - Encourage programmers to think about and make manifest their assumptions

Oops — We Can't Do This!

- Rice's Theorem: No computer program can precisely determine anything interesting about arbitrary source code
 - Does this program terminate?
 - Does this program produce value 42?
 - Does this program raise an exception?
 - Is this program correct?

The Art of Static Analysis

- Programmers don't write arbitrarily complicated programs
- Programmers have ways to control complexity
 - Otherwise they couldn't make sense of them
- Target: Be precise for the programs that programmers want to write
 - It's OK to forbid yucky code in the name of safety

Research at the University of Maryland

- Developed a number of practical tools addressing different software quality issues
 - CQual — User-defined type qualifiers for C
 - Locksmith — C data race detection
 - FindBugs — Finding (Java) bugs is easy
 - Cyclone — Language for safe, low-level programming
 - Ginseng — Safe updates to running software
 - Saffire — Type checking multi-lingual programs
 - Pistachio — Checking network protocol implementations

CQual: Background

- Tools need specifications

```
spin_lock_irqsave(&tty->read_lock, flags);
```

```
put_tty_queue_nolock(c, tty);
```

```
spin_unlock_irqrestore(&tty->read_lock, flags);
```

- Goal: Add specifications to programs

In a way that...

- Programmers will accept
 - Lightweight
- Scales to large programs
- Solves many different problems

Type Qualifiers

- Extend standard type systems (C, Java, ML)
 - Programmers already use types
 - Programmers understand types
 - Get programmers to write down a little more...

`const int`

ANSI C

`ptr(tainted char)`

Format-string vulnerabilities

`kernel ptr(char) → char`

User/kernel vulnerabilities

Application: Format String Vulnerabilities

- I/O functions in C use format strings

```
printf("Hello!");
```

Hello!

```
printf("Hello, %s!", name);
```

Hello, name!

- Instead of

```
printf("%s", name);
```

Why not

```
printf(name);
```

?

Format String Attacks

- Adversary-controlled format specifier

```
name := <data-from-network>
printf(name); /* Oops */
```

 - Attacker sets name = "%s%s%s" to crash program
 - Attacker sets name = "...%n..." to write to memory
 - Yields (often remote root) exploits
- Lots of these bugs in the wild
 - New ones weekly on bugtraq mailing list
 - Too restrictive to forbid variable format strings

Using Tainted and Untainted

- Add qualifier annotations

```
int printf(untainted char *fmt, ...)
```

```
tainted char *getenv(const char *)
```

tainted = may be controlled by adversary

untainted = must not be controlled by adversary

Subtyping

```
void f(tainted int);  
untainted int a;  
f(a);
```

OK

f accepts tainted or
untainted data

untainted \leq tainted

```
void g(untainted int);  
tainted int b;  
f(b);
```

Error

g accepts only untainted
data

tainted $\not\leq$ untainted

untainted $<$ tainted

Demo of cqual

<http://cqual.sourceforge.net>

Type Qualifier Inference

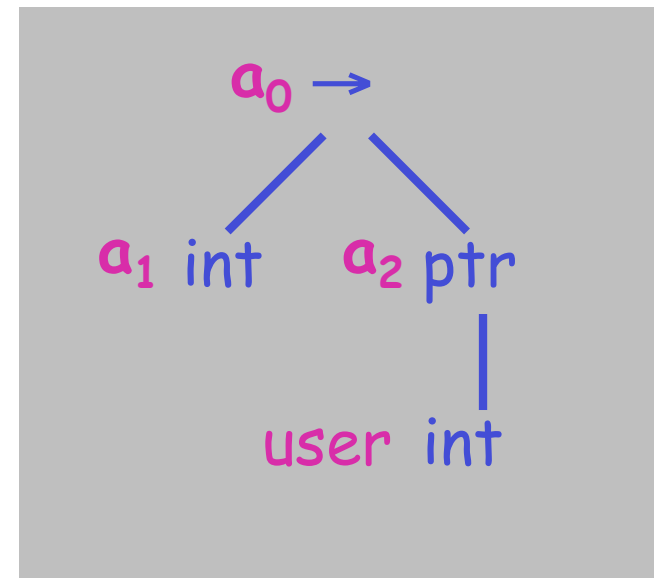
- Two kinds of qualifiers
 - Explicit qualifiers: **tainted**, **untainted**, ...
 - Unknown qualifiers: **a₀**, **a₁**, ...
- Program yields constraints on qualifiers
tainted ≤ **a₀** **a₀** ≤ **untainted**
- Solve constraints for unknown qualifiers
 - Error if no solution

Types as Trees

ptr(tainted char)



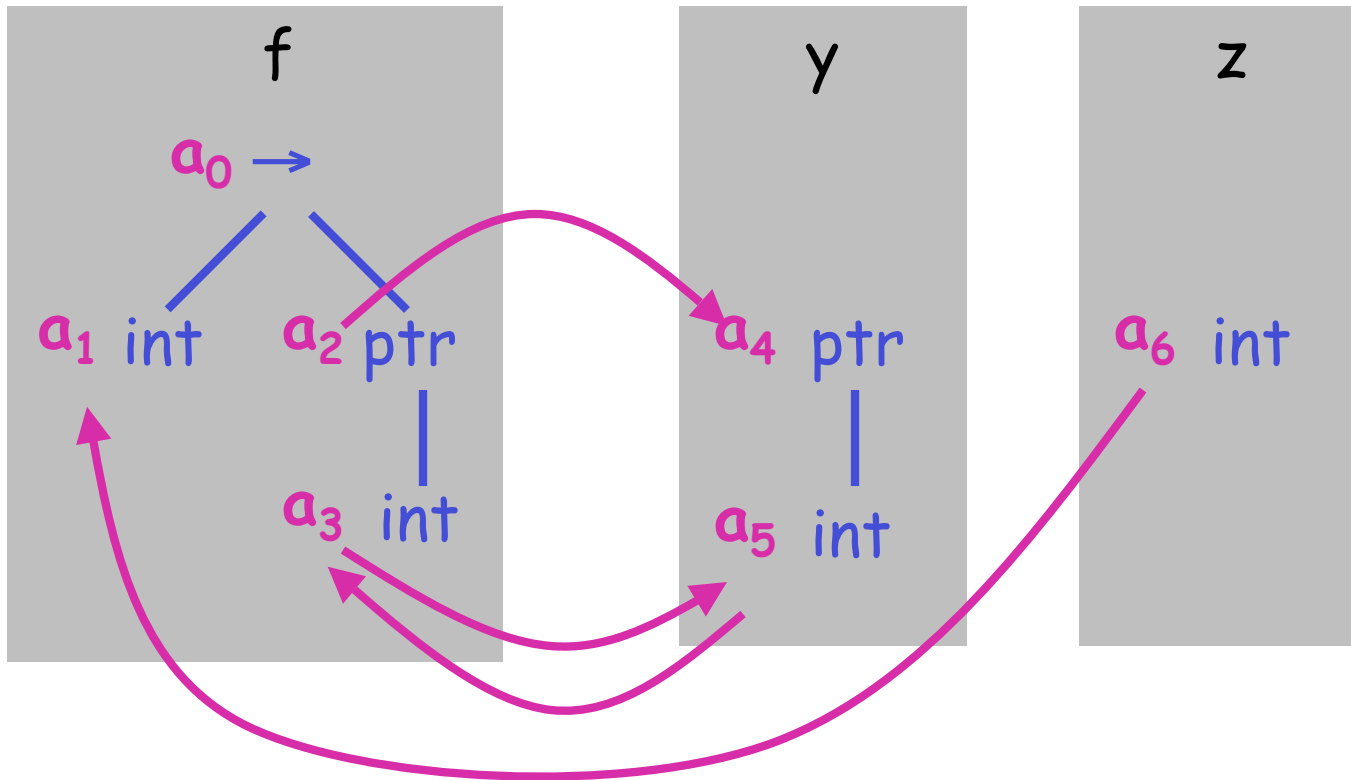
int \rightarrow user ptr(int)



Constraint Generation

ptr(int) f(x : int) = { ... }

y := f(z)

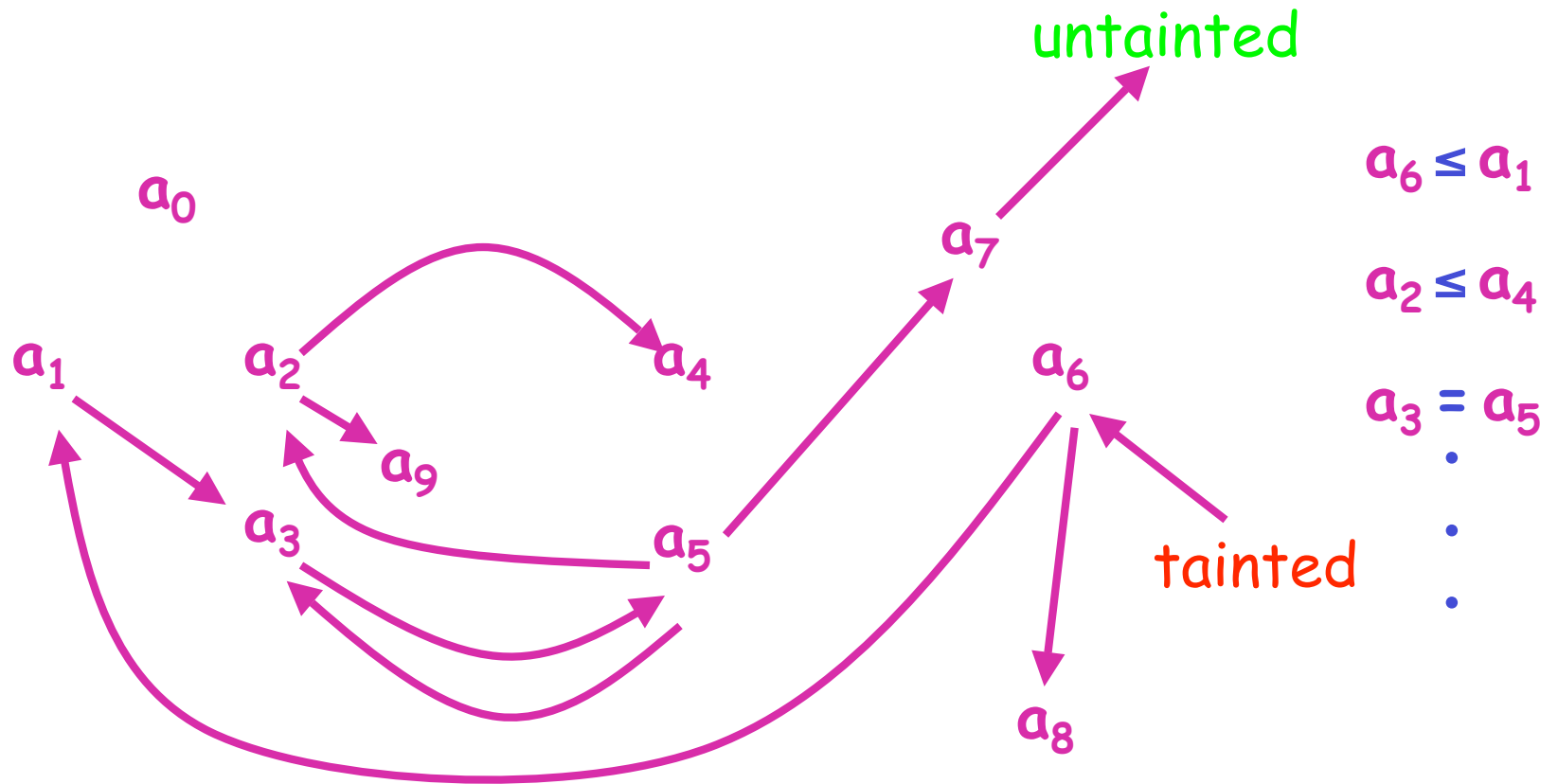


$$a_6 \leq a_1$$

$$a_2 \leq a_4$$

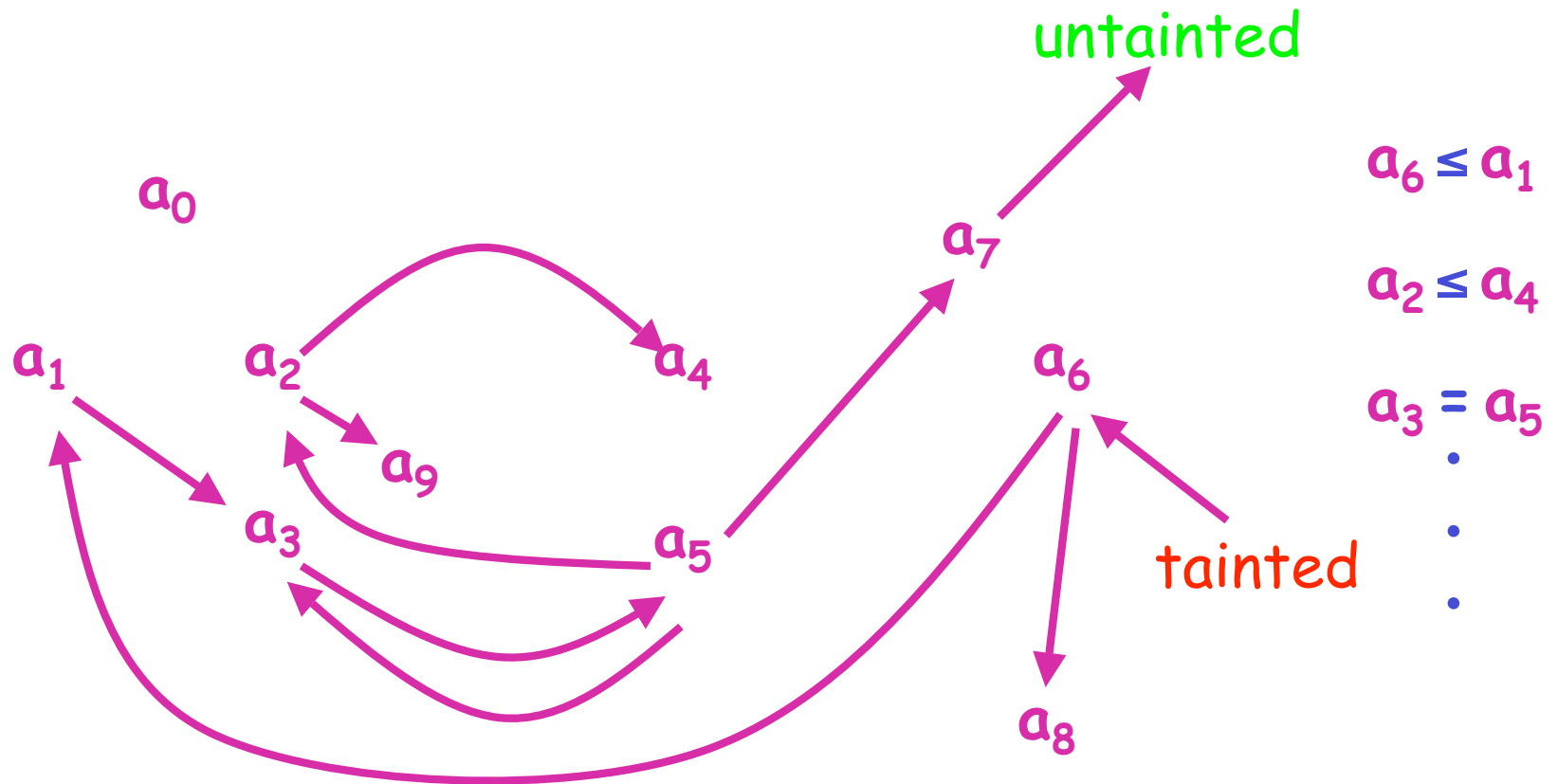
$$a_3 = a_5$$

Constraints as Graphs



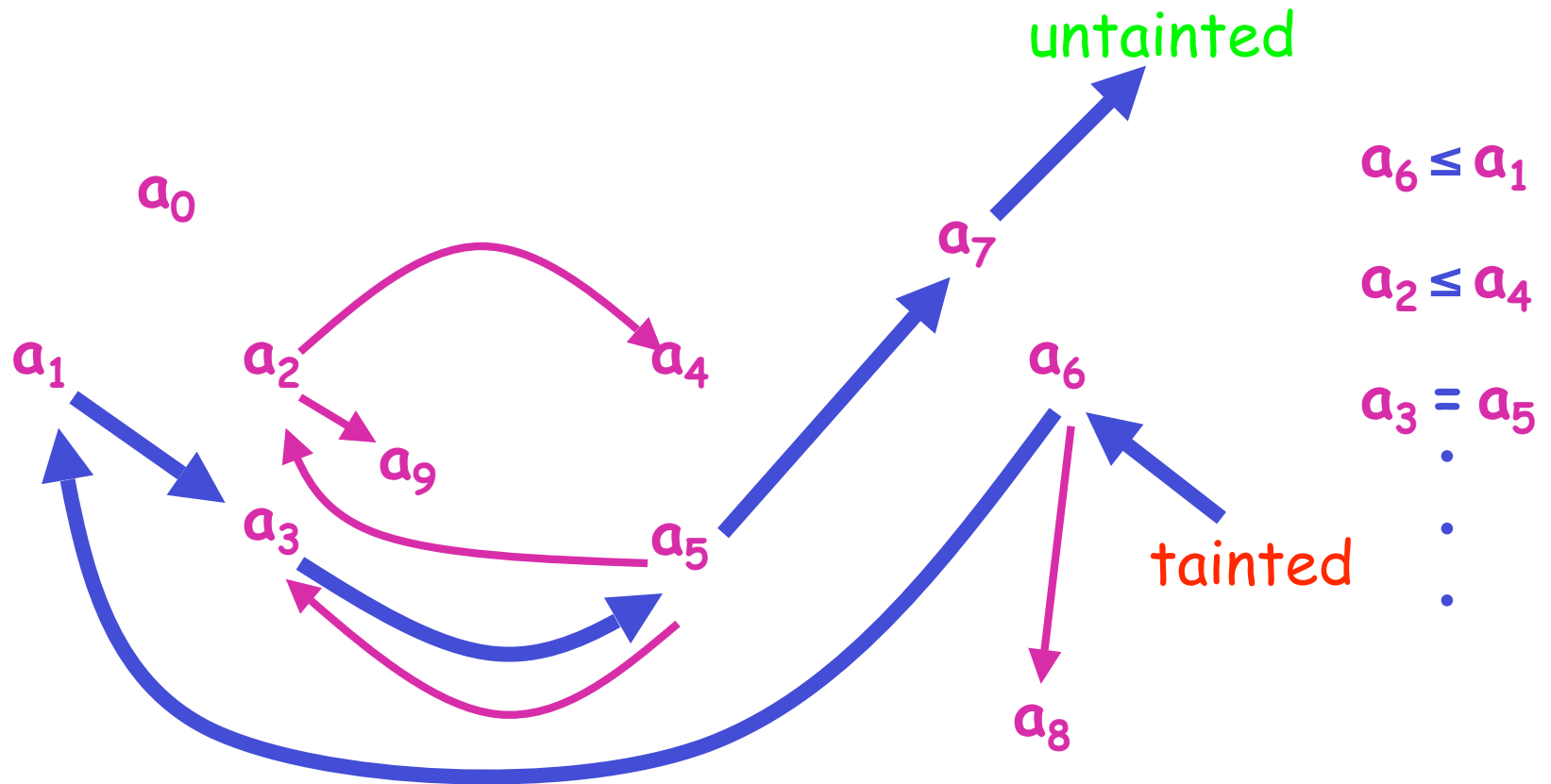
Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



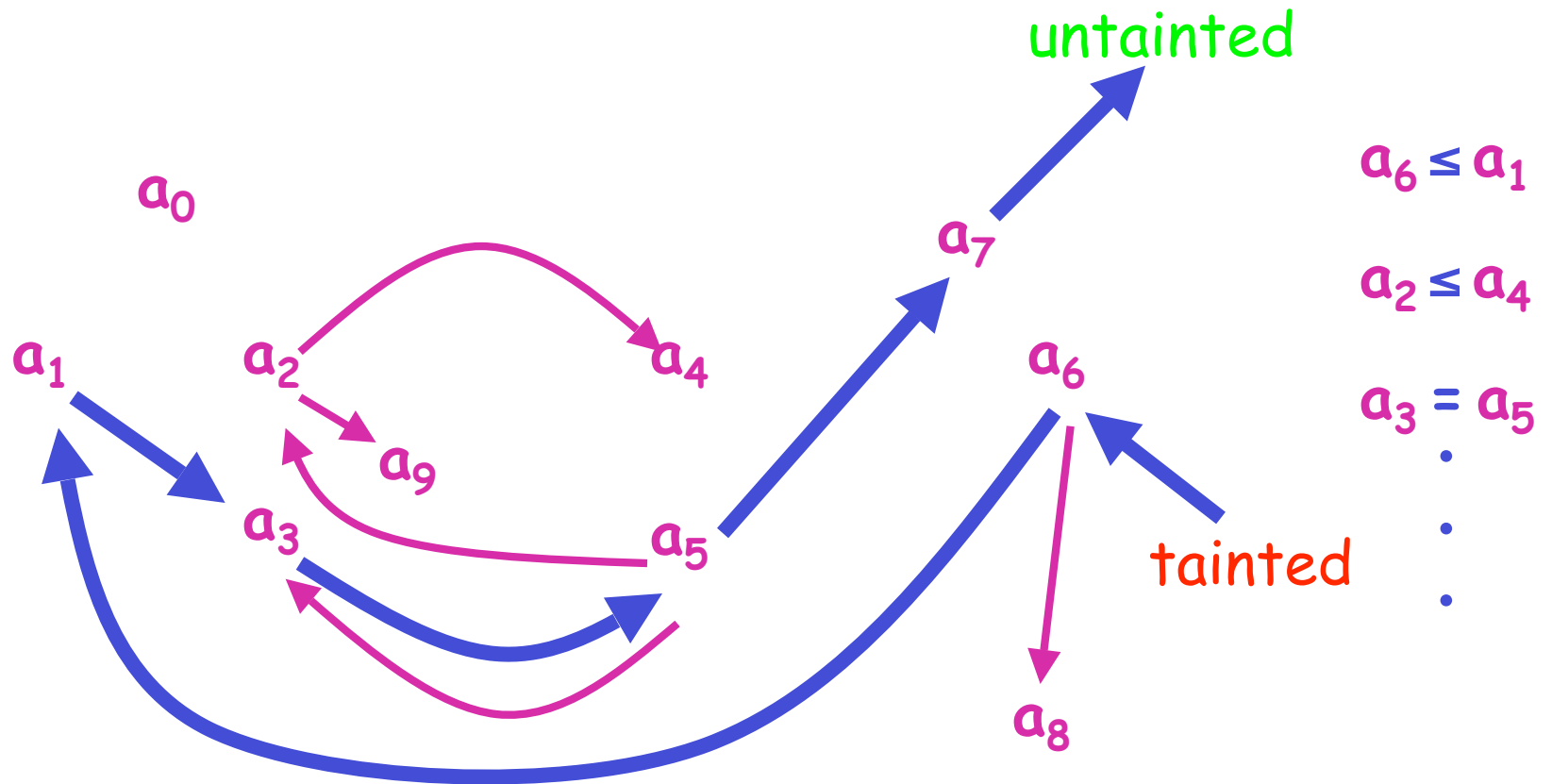
Satisfiability via Graph Reachability

Is there an inconsistent path through the graph?



Satisfiability via Graph Reachability

tainted \leq $a_6 \leq a_1 \leq a_3 \leq a_5 \leq a_7 \leq$ untainted



Satisfiability in Linear Time

- Initial program of size n
 - Fixed set of qualifiers **tainted**, **untainted**, ...
- Constraint generation yields $O(n)$ constraints
 - Recursive abstract syntax tree walk
- Graph reachability takes $O(n)$ time
 - Works for semi-lattices, discrete p.o., products

Experiment: Format String Vulnerabilities

- Analyzed 10 popular unix daemon programs
 - Annotations shared across applications
 - One annotated header file for standard libraries
 - Includes annotations for polymorphism
 - Critical to practical usability
- Found several known vulnerabilities
 - Including ones we didn't know about
- User interface critical

Results: Format String Vulnerabilities

Name	Warn	Bugs
identd-1.0.0	0	0
mingetty-0.9.4	0	0
bftpd-1.0.11	1	1
muh-2.05d	2	~2
cfengine-1.5.4	5	3
imapd-4.7c	0	0
ipopd-4.7c	0	0
mars_nwe-0.99	0	0
apache-1.3.12	0	0
openssh-2.3.0p1	0	0

Other CQual Applications

- User/kernel pointer vulnerabilities
 - Common security bug in the Linux kernel
- Deadlock in the Linux kernel
- File open/close
 - Both require flow-sensitivity
- Initialization in the Linux kernel
- **const** inference

CQual — Summary

- Type qualifiers are specifications that...
 - Programmers will accept
 - Lightweight
 - Scale to large programs
 - Solve many different problems
- In the works: `ccqual`, `jqual`, Eclipse interface

Data Races

- Two threads access the same location simultaneously (and one is a write)
- 2 of the "top ten bugs of all time" due to races
 - 2003 Northeastern US blackout
 - 1985-87 Therac-25 medical accelerator
- Data races complicate program understanding

Programming Against Races

- Correlation $\rho \otimes l$:
 - "Lock l is *correlated* with location ρ if lock l is held whenever ρ is dereferenced
- Consistent Correlation
 - Each location ρ is always correlated with the same lock(s)
- Sometimes say that ρ is *guarded by* l

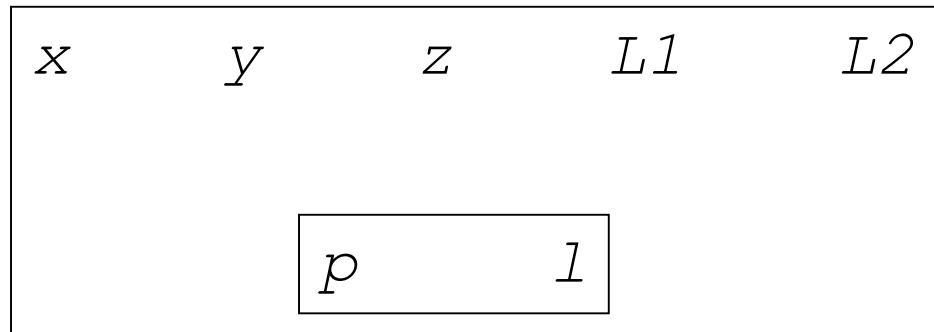
Inferring consistent correlation

- **Locksmith** is a static analysis tool we have built to discover consistent correlation statically for C programs
- Contradictions of consistent correlation are possible races
- *Sound*: applies to all possible program executions

Correlation

```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
foo (&L1, &x) ;  
foo (&L2, &y) ;  
foo (&L2, &z) ;
```

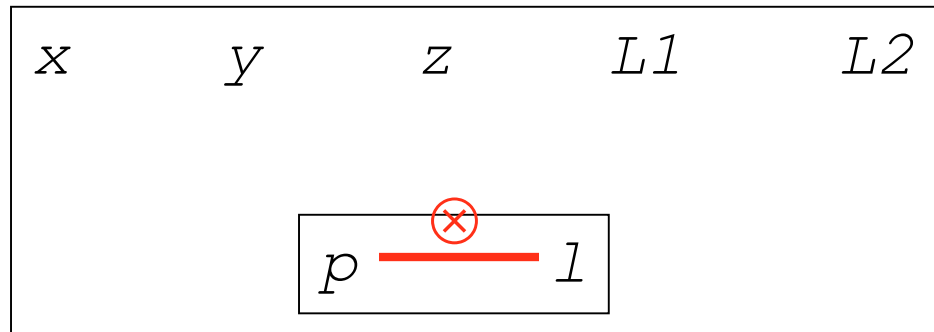


Correlation

```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

...

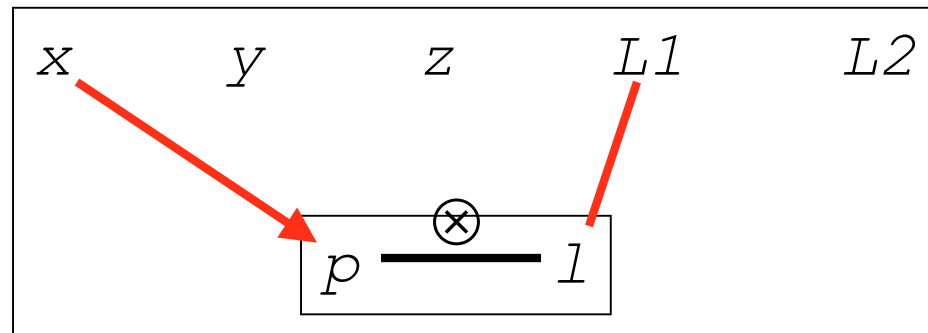
```
foo (&L1, &x) ;  
foo (&L2, &y) ;  
foo (&L2, &z) ;
```



Correlation

```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l),  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

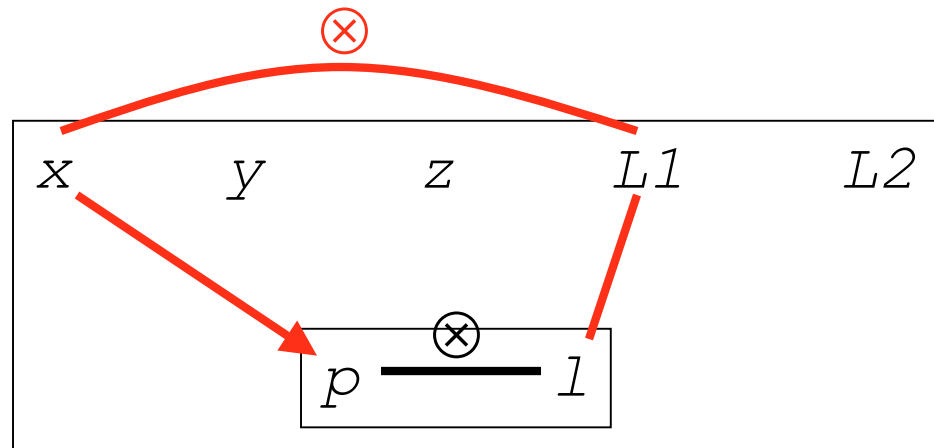
```
...  
foo (&L1, &x) ;  
foo (&L2, &y) ;  
foo (&L2, &z) ;
```



Correlation

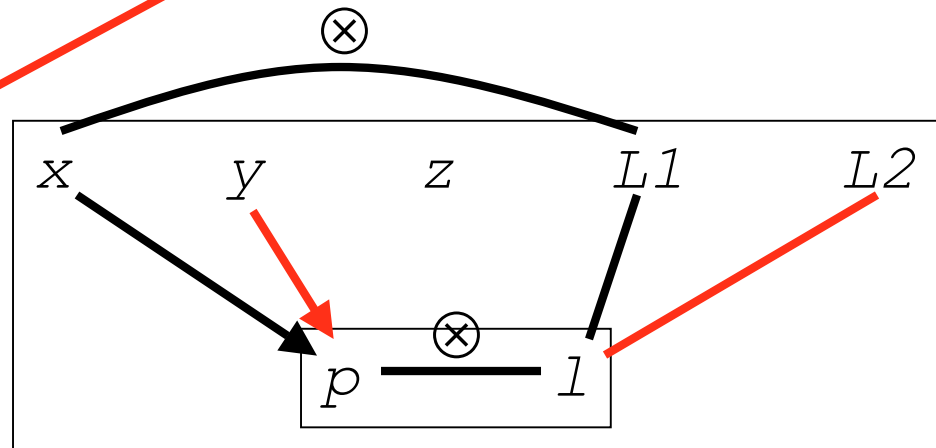
```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
foo (&L1, &x) ;  
foo (&L2, &y) ;  
foo (&L2, &z) ;
```



Correlation

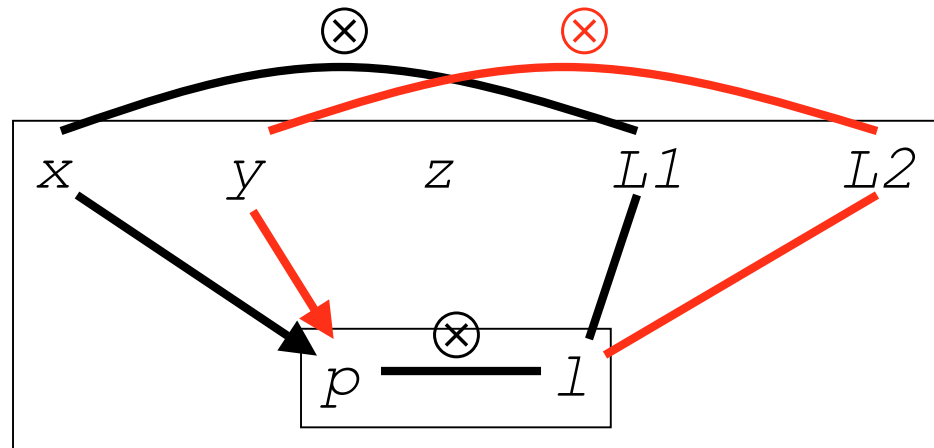
```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}  
...  
foo (&L1, &x);  
foo (&L2, &y);  
foo (&L2, &z);
```



Correlation

```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

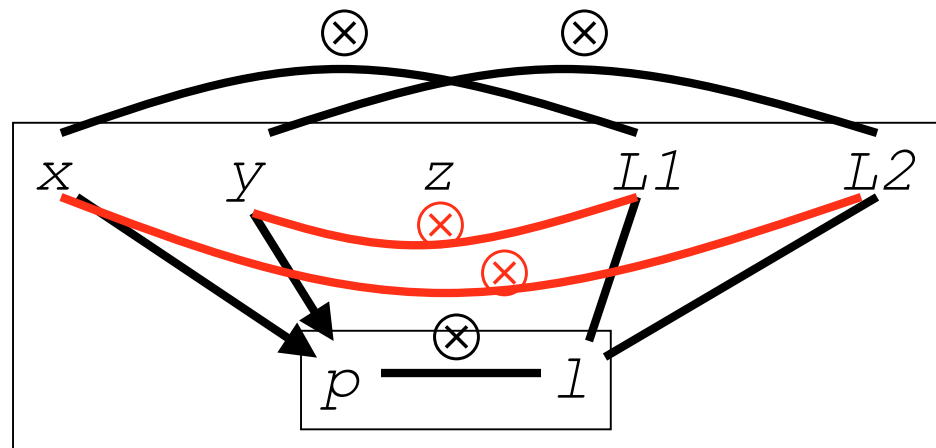
```
...  
foo (&L1, &x) ;  
foo (&L2, &y) ;  
foo (&L2, &z) ;
```



Correlation

```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
foo (&L1, &x) ;  
foo (&L2, &y) ;  
foo (&L2, &z) ;
```



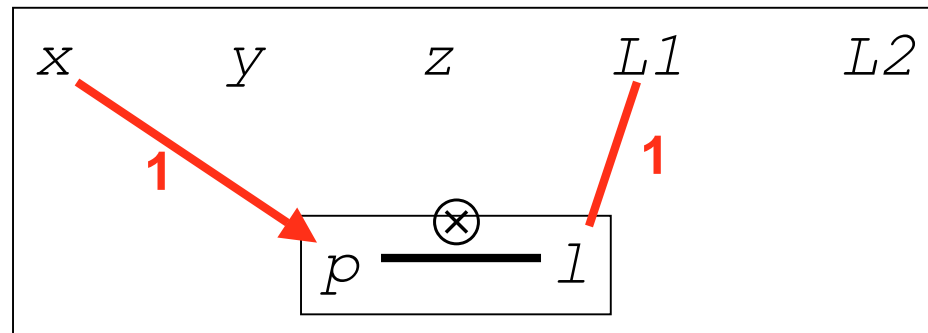
Context Sensitivity

- The problem arises because within `foo()` we were not distinguishing different call sites
- Solution: *context sensitivity*
 - label each call site
 - label edges induced by that call site
 - propagate constraints along like-labeled paths

Correlation

```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l),  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

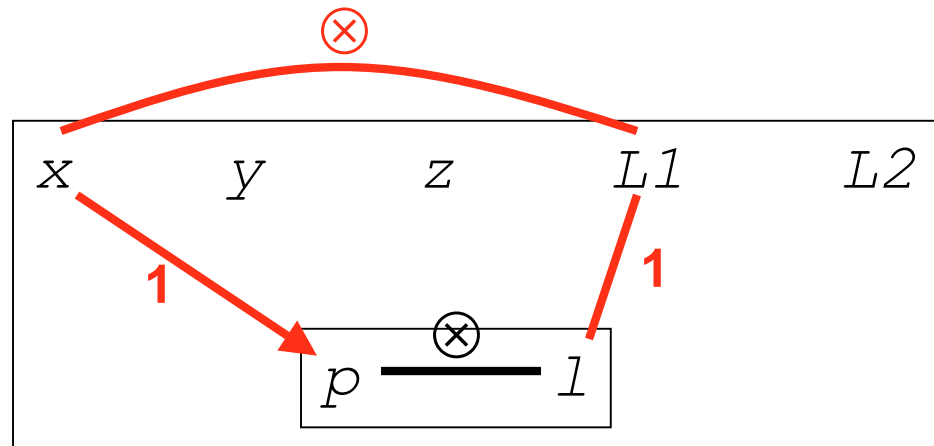
```
...  
foo1 (&L1, &x);  
foo2 (&L2, &y);  
foo3 (&L2, &z);
```



Correlation

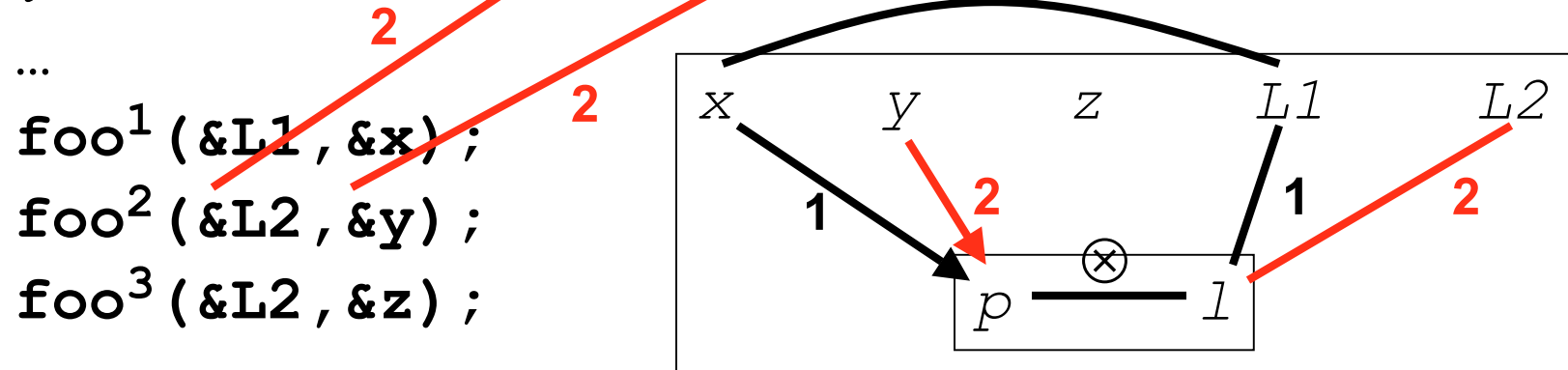
```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
foo1(&L1, &x);  
foo2(&L2, &y);  
foo3(&L2, &z);
```



Correlation

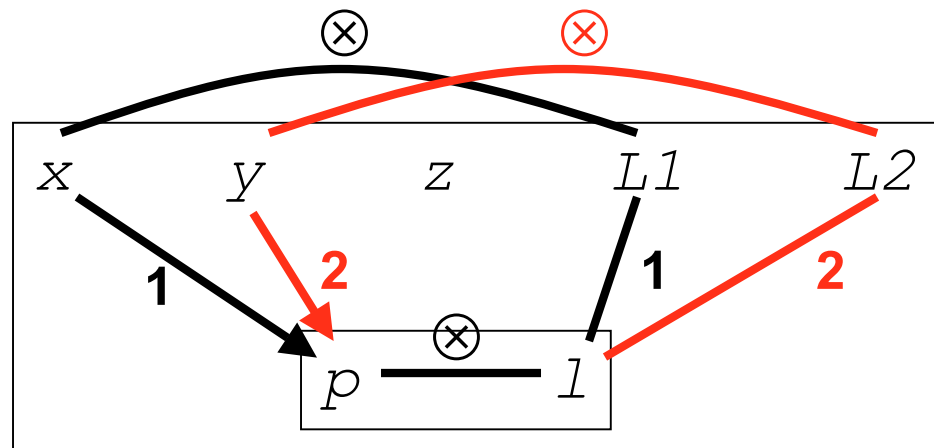
```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```



Correlation

```
pthread_mutex_t L1, L2;  
int x,y,z;  
void foo(pthread_mutex_t *l, int *p) {  
    pthread_mutex_lock(l);  
    *p = 3;  
    pthread_mutex_unlock(l);  
}
```

```
...  
foo1(&L1, &x);  
foo2(&L2, &y);  
foo3(&L2, &z);
```



Soundness

- Formalized correlation inference in a small formal language
- Proved that if constraints induced by the program have a solution, the program is consistently correlated
- May have other applications
 - Correlating a pointer with the region it points to
 - Correlation an array with its length

Scaling to C: Challenges

- Aliasing data structures containing locks
 - Danger of not identifying locks precisely
- Determining which locks are held
 - Which locks are held is a flow-sensitive, interprocedural property
- Determining which locations are shared
 - A flow-sensitive property
- Type-unsafe idioms
 - Casts to/from void*
- Scaling to large data structures

Locks in data structures

```
struct foo {  
    pthread_mutex_t l;  
    int *data;  
    struct foo *next;  
};
```

- Alias analysis cannot distinguish individual elements
- Want to precisely reason about correlation *within* an particular element.

Locks in data structures

```
struct foo {  $\exists l, \rho. \rho \otimes l$   
    pthread_mutex_t  $\langle l \rangle$  l;  
    int * $\rho$  data;  
    struct foo *next;  
};
```

- Alias analysis cannot distinguish individual elements
- Want to precisely reason about correlation *within* an particular element.

Experimental Results

- Ran Locksmith on a series of benchmarks
 - Standalone POSIX threads programs
 - Linux device drivers
- Measurements on a dual core Xeon with 4 GB of RAM.

Standalone Programs

Program	Size (KLOC)	Time	Warn.	Unguarded	Races
aget	1.6	0.8s	15	15	15
ctrace	1.8	0.9s	8	8	2
pfscan	1.7	0.7s	5	0	0
engine	1.5	1.2s	7	0	0
smtprc	6.1	6.0s	46	1	1
knot	1.7	1.5s	12	8	8

Device Drivers

Driver	Size (KLOC)	Time	Warn.	Unguarded	Races
plip	19.1	24.9s	11	2	1
eq1	16.5	3.2s	3	0	0
3c501	17.4	240.1s	24	2	2
sundance	19.9	98.2s	3	1	0
sis900	20.4	61.0s*	8	2	1
slip	22.7	16.5s*	19	1	0
hp100	20.3	31.8s*	23	2	0

(*) Run without lock linearity analysis

Summary

- Discover races automatically by inferring consistent correlation
- Formalized and proved correct for a core language
- Implemented a tool for C programs
 - Available at <http://www.cs.umd.edu/~polyvios/locksmith>

Other Tools We've Built

- Cyclone — Language for safe, low-level programming
- FindBugs — Finding (Java) bugs is easy
- Ginseng — Safe updates to running software
- Saffire — Type checking multi-lingual programs
- Pistachio — Checking network protocol implementations

Cyclone: Safe Low-Level Programming

- Today, our economy, government, and military depend upon the proper functioning of our computing and communications infrastructure.
- That infrastructure is coded in low-level, error-prone languages (*i.e.*, C).
 - device drivers, kernels
 - file systems, web servers, email systems
 - switches, routers, firewalls

Cyclone: An Experimental Safe-C

- Start with ANSI-C
- Throw out anything that can lead to a delayed core-dump:
 - Arbitrary casts, unchecked pointer arithmetic, etc.
- Add advanced typing mechanisms and dynamic checks to cover what's missing
 - Programmer will have to specify additional details at procedure boundaries
- Minimize re-coding for safe idioms.
 - Best case: leave the code alone
 - Next best: add typing annotations
 - Worst case: re-write the code

Unifying Theme: Region types

- Conceptually divide memory into **regions**
 - Different kinds of regions (e.g., not just bulk-free)
- Associate every pointer with a region
- Prevent dereferencing pointers into dead regions

```
int *`r x; // x points into region `r  
*x = 3;    // deref allowed if `r is live
```

(inference often obviates annotations `r)
Liveness by static analysis

LIFO Arenas

- Dynamic allocation mechanism
- Lifetime of entire arena is scoped
 - At conclusion of scope, all data allocated in the arena is freed.

LIFO Arena Example

```
FILE *infile = ...
Image *i;
if (tag(infile) == HUFFMAN) {
    region<`r> h; // region `r created
    struct hnode *`r huff_tree;
    huff_tree = read_tree(h,infile); //
    allocates with h
    i = decode_image(infile,huff_tree,...);
    // region `r deallocated upon exit of scope
} else ...
```

Regions Summary

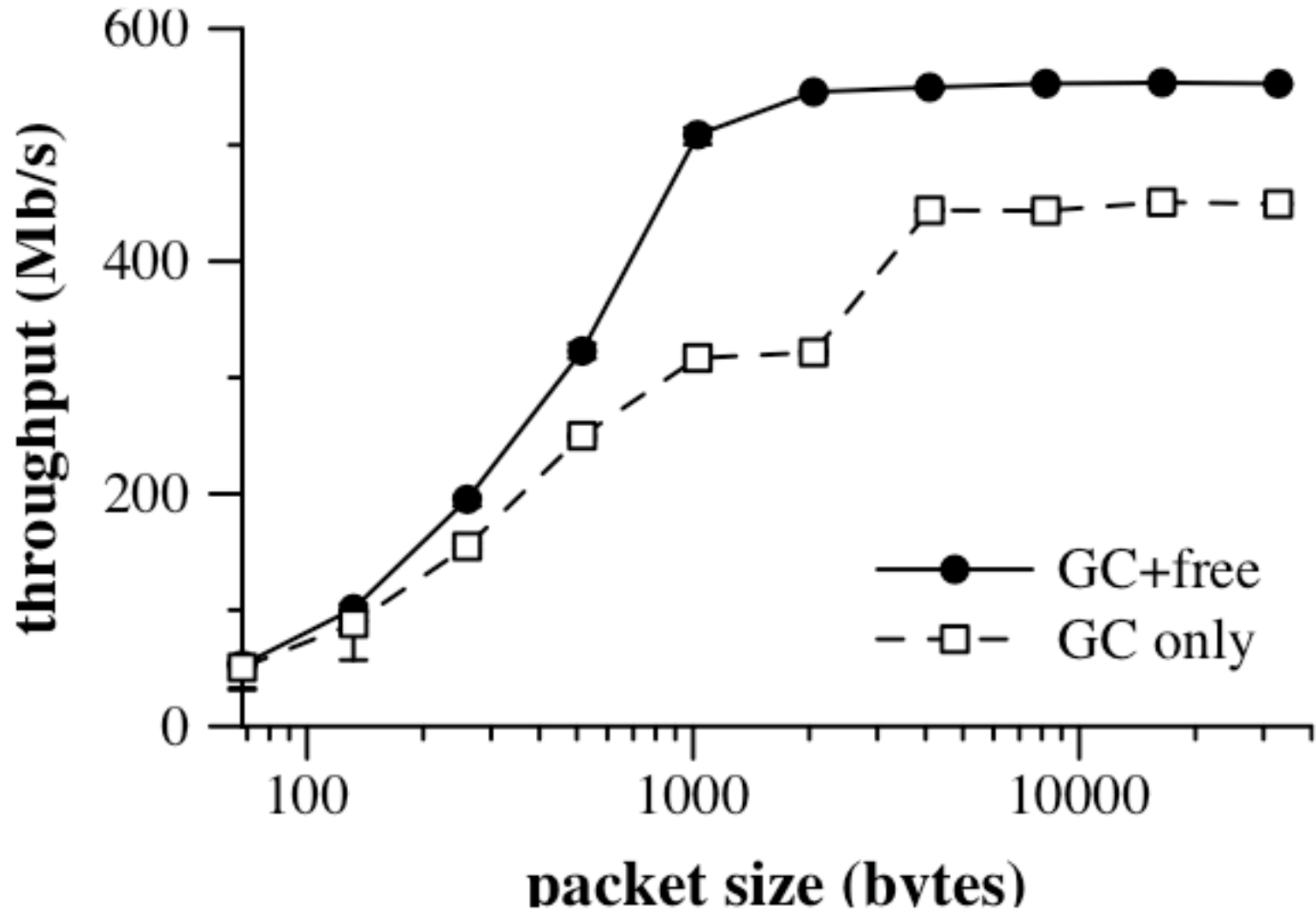
Region Variety	Allocation (objects)	Deallocation (what) (when)		Aliasing (objects)
Stack	static	whole region	exit of scope	free
LIFO	dynamic		manual	
Dynamic		single objects	GC	
Heap			manual	restricted
Unique				
Refcounted				

Application Characteristics

Program	Non-comment Lines of Code			Manual mechs
	C	Cyc	Cyc (+manual)	
Boa	5217	± 284 (5%)	± 91 (1%)	U(D)
BetaFTPD	1146	± 191 (16%)	± 225 (21%)	UR
Epic	2123	± 217 (10%)	± 114 (5%)	UL
Kiss-FFT	453	± 73 (16%)	± 20 (4%)	U
MediaNet		8715	± 320 (4%)	URLD
CycWeb			667	U
CycScheme			2523	ULD

U = unique pointers R = ref-counted pointers
 L = LIFO regions D = dynamic arenas

Throughput: MediaNet



Conclusions

- *High degree of control, safely:*
- Sound mechanisms for programmer-controlled memory management
 - Region-based and object-based deallocation
 - Manual and automatic reclamation
- Region-annotated pointers within a simple framework
 - Scoped regions unifying theme (alias, open)
 - Region polymorphism, for code reuse

FindBugs

- Programmers make lots of dumb mistakes in their code
 - Dereferencing a pointer that's obviously null
 - Writing an infinite loop
 - Ignoring the return value of a method
 - Assigning a field to itself
- Idea: Develop *bug pattern detectors* to find coding styles that are certain to be, or likely to be, bugs



Results: Infinite Loops

```
private final String foundType;  
public String foundType() {  
    return this.foundType();  
}
```

- 5 Sun's JDK 1.5.0_01
- 10 Sun's AppServer 8.1 2005Q1
- 14 IBM's WebSphere 6.0.2
- 13 JBoss 4.0.2
- 3 Eclipse 3.1M7
- 2 Tomcat 5.5.9

Results: Null Pointers

```
public String getContentId()
{
    if (header != null || header.length > 0 )
        ...
}
```

- 43 Sun's JDK 1.5.0_01
- 199 Sun's AppServer 8.1 2005Q1
- 465 IBM's WebSphere 6.0.2
- 108 JBoss 4.0.2
- 90 Eclipse 3.1M7
- 16 Tomcat 5.5.9

Program Analysis in Practice

- We've described a number of practical tools
 - At least, *we* can effectively use them
 - Several have also been used by others
 - E.g., >100,000 downloads of FindBugs
- Sophisticated program analyses are moving into practice
 - Microsoft static driver verifier
 - Microsoft PREFIX/PREFast
 - Coverity
 - Fortify

Research Challenge: Explaining the Analyses

- When a tool reports a potential problem, how do we explain it to the user?
 - less than 10% of the "code mass" of PREFIX is the actual program analysis (Pincus)
- What our tools do
 - CQual: Show a path through the graph
 - Locksmith: Show two paths through the graph, plus some other stuff (e.g., locks held)

Limitations of Current Explanations

- In our experience, a path often conveys enough information
 - E.g., the earlier CQual demo
- However, in some cases paths are confusing
 - Paths may go in and out of functions
 - And our analysis is context-sensitive
 - Aliasing can cause non-intuitive "backward" flow
 - Paths may be correlated
 - Acquiring locks vs. data flow of accessed locations

The Abstraction Barrier

- In order to attack an undecidable problem (program analysis), we needed to make an approximation
 - We performed a kind of abstract interpretation of the program
- When the abstraction matches the programmer's intuition, then it's great
- But when the programmer doesn't understand the abstraction, explanations are hard

User-Centered Program Analysis

- The field of program analysis has made great strides in recent years
- We believe we need new techniques that are user-centered
 - We need to be better at integration into the software engineering process
 - We need to match programmer intuitions
 - But we also need to train programmers that a little effort on their part (e.g., annotations, not coding in complex ways) can gain them benefits

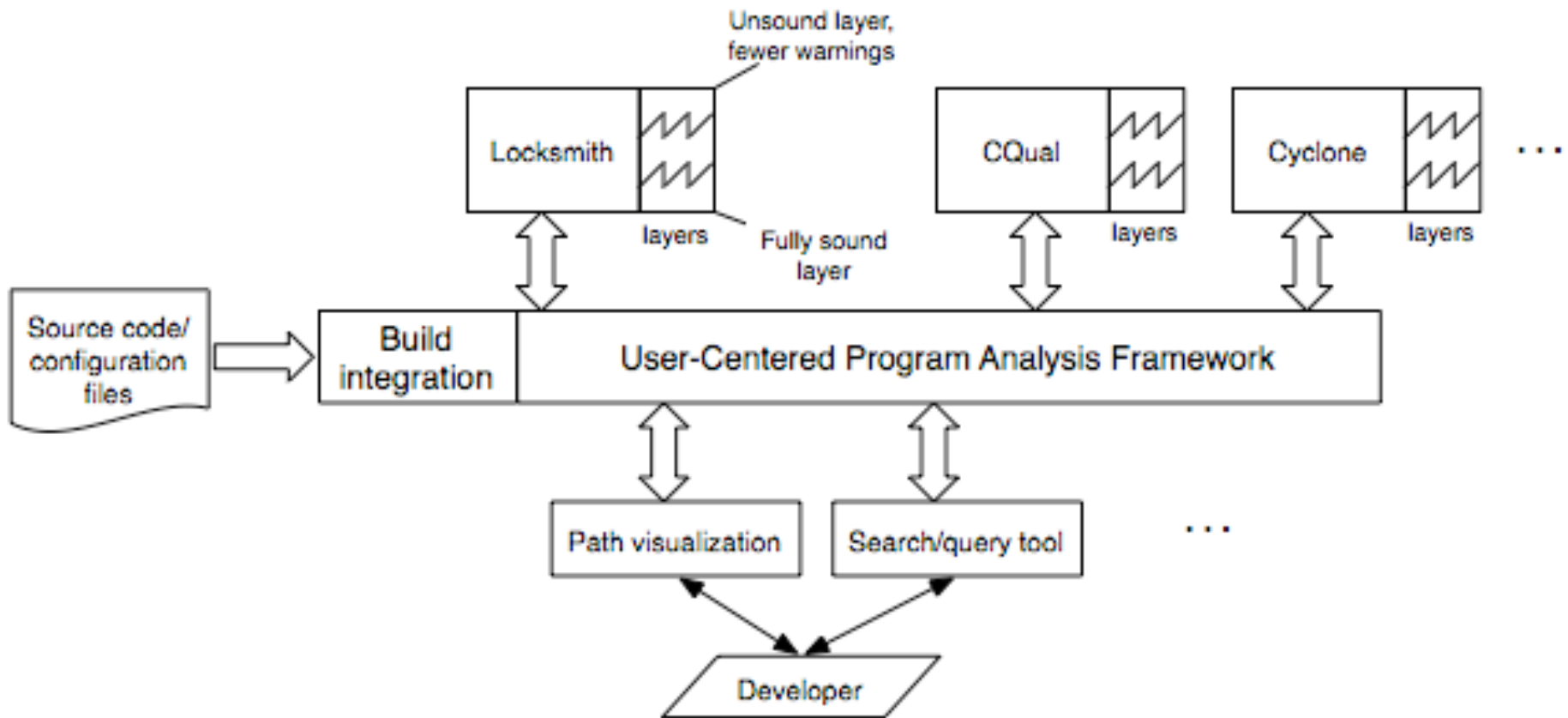
Multi-level analysis

- Low threshold, high ceiling
 - Tool should be initially easy to use and understand
 - But should scale up to more sophisticated users
- Multiple levels: vary a parameter to aid programmer understanding
- Example: varying soundness
 - Simplest mode: all alarms certain to be errors
 - Advanced mode: all emitted warnings cover all possible errors

Finding Bugs, Fixing Bugs

- The tool develops a model of the program
 - A contradiction suggests a bug
- The programmer must determine
 - Is the bug for real?
 - How do I fix it?
- Idea: provide access to the model in terms that the programmer can understand
 - Queries/filters in terms of program text
 - More direct than generic search (e.g., grep)

One Possible Framework



Evaluation Techniques

- User studies
 - Control well for variables as users carry out specific tasks using our program analysis tools
- Pluses
 - Reliable scientific technique
 - Can measure length of time to solve problem etc
- Minuses
 - Requires tool to be fixed throughout experiment
 - Hard to carry out long-term study
 - State of the art might change during the study!

Evaluation Techniques (cont'd)

- Multi-dimensional in-depth longitudinal case studies
 - A "soft" technique where we observe programmers over a long period of time
- Plusses
 - Lets the tools evolve as part of the experimental process
 - Longitudinal input about tools from users
- Cons
 - Cannot control variables well; hard to perform quantitative measurements

Summary

- Software is buggy
- Bugs can lead to crashes and security vulnerabilities
- Static analysis tools can help
 - find bugs
 - prove the absence of certain flaws
 - programmers understand software
- At Maryland, we have developed a variety of tools, and continue to push the state of the art to make them more effective

For More Information

<http://www.cs.umd.edu/projects/PL>

Code, papers, and more.