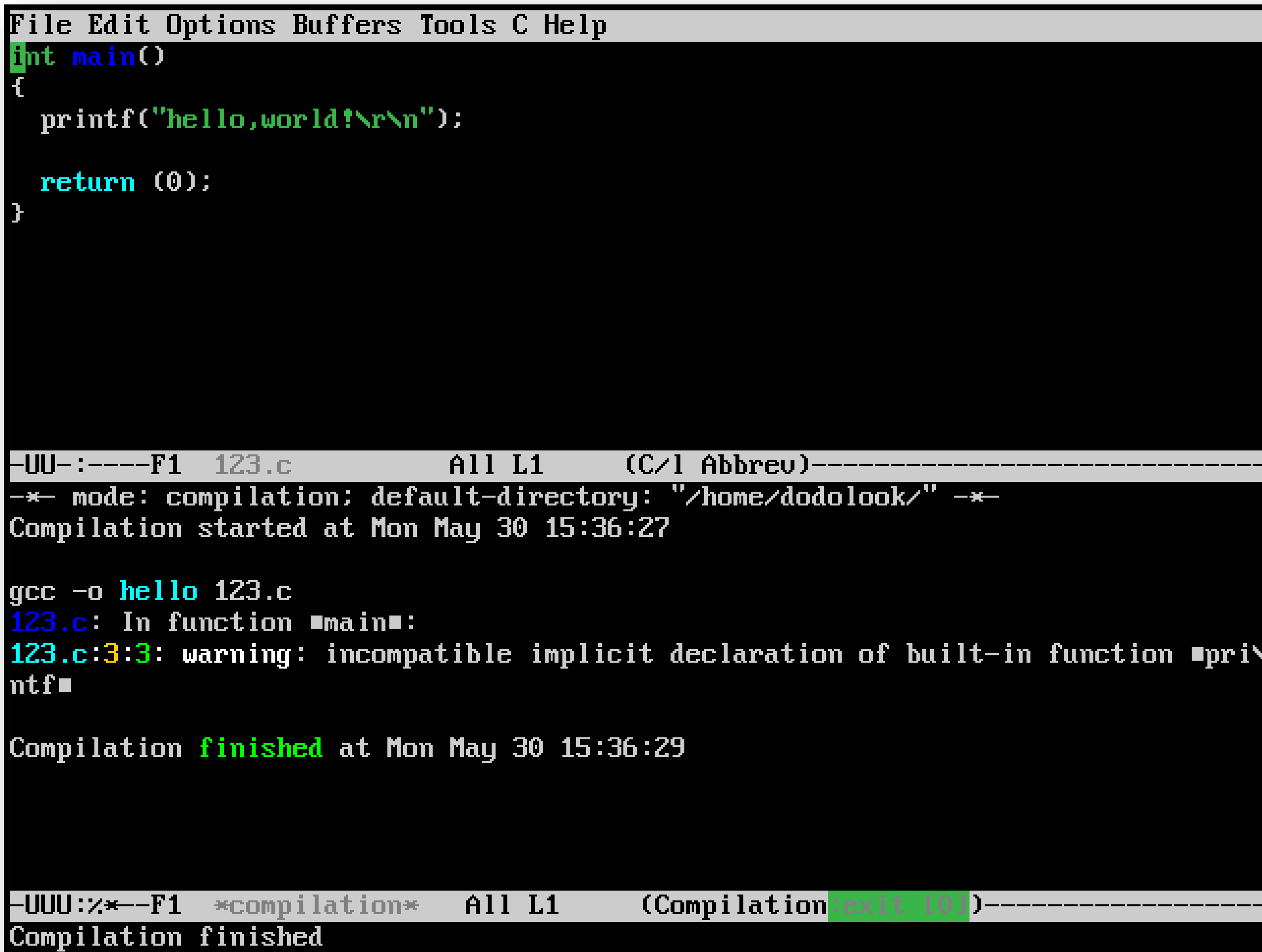
SERVER-SIDE ANALYSIS

Ben Livshits, Microsoft Research

Overview of Today's Lecture

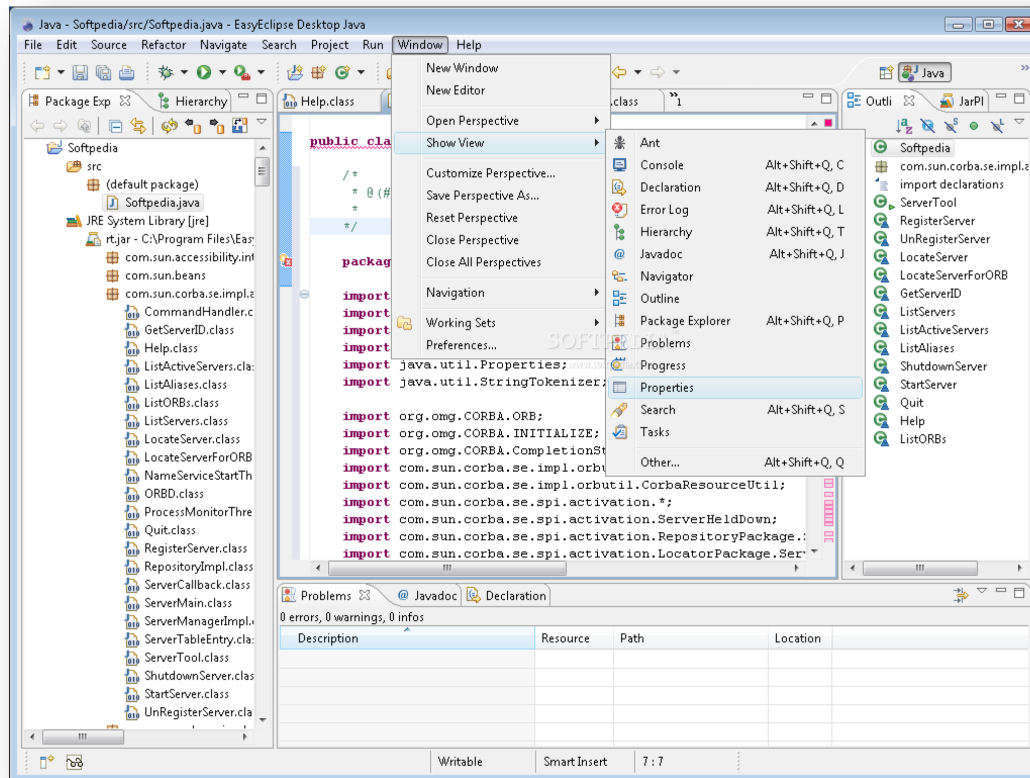
- Static analysis for bug finding
- Scripting languages analyzed (UsenixSec '05 paper)
- Runtime analysis
  - Fuzzing
  - Pen testing
  - Tainting
  - Symbolic execution

Compilers Under the Hood



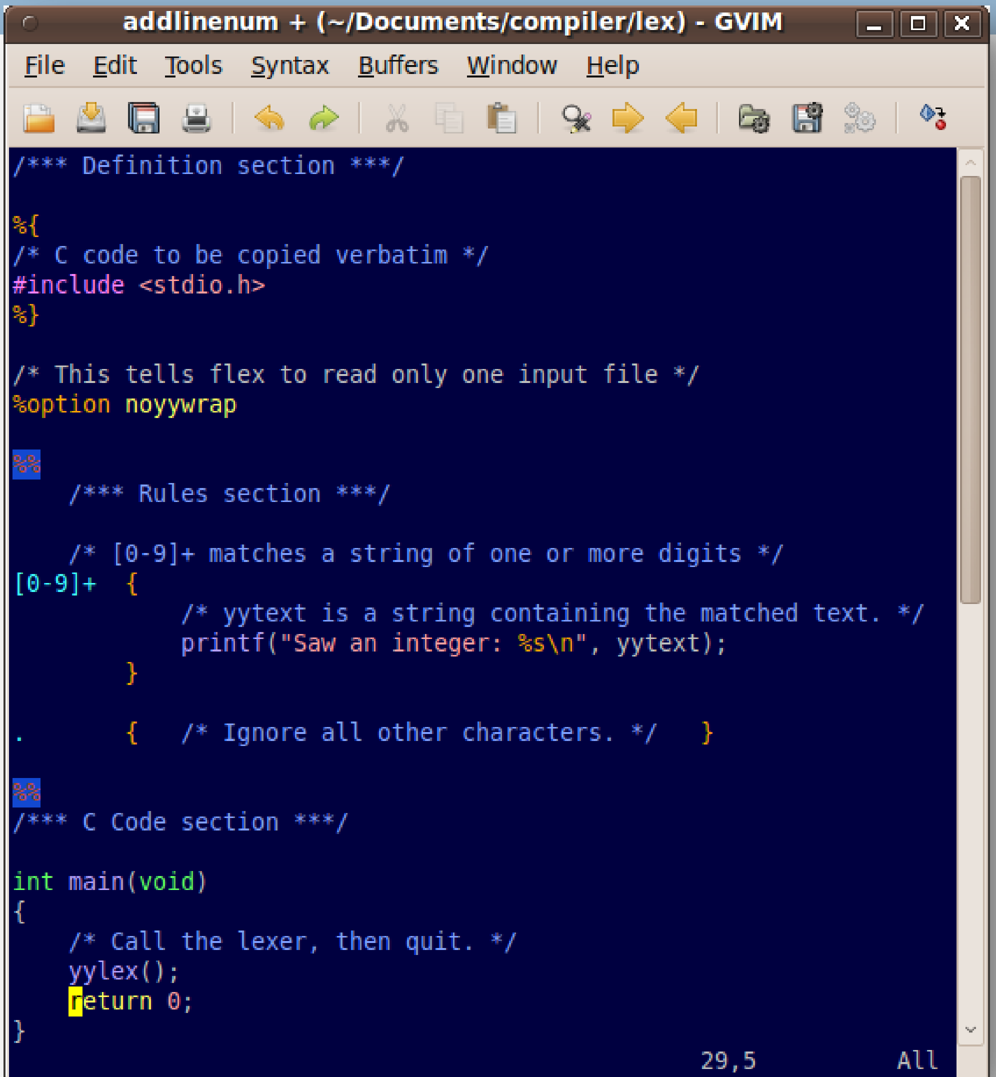
Stages of Compilation

- Source code
- Lexing
- Parsing
- IR
- Analysis
- Code generation
- Executable code



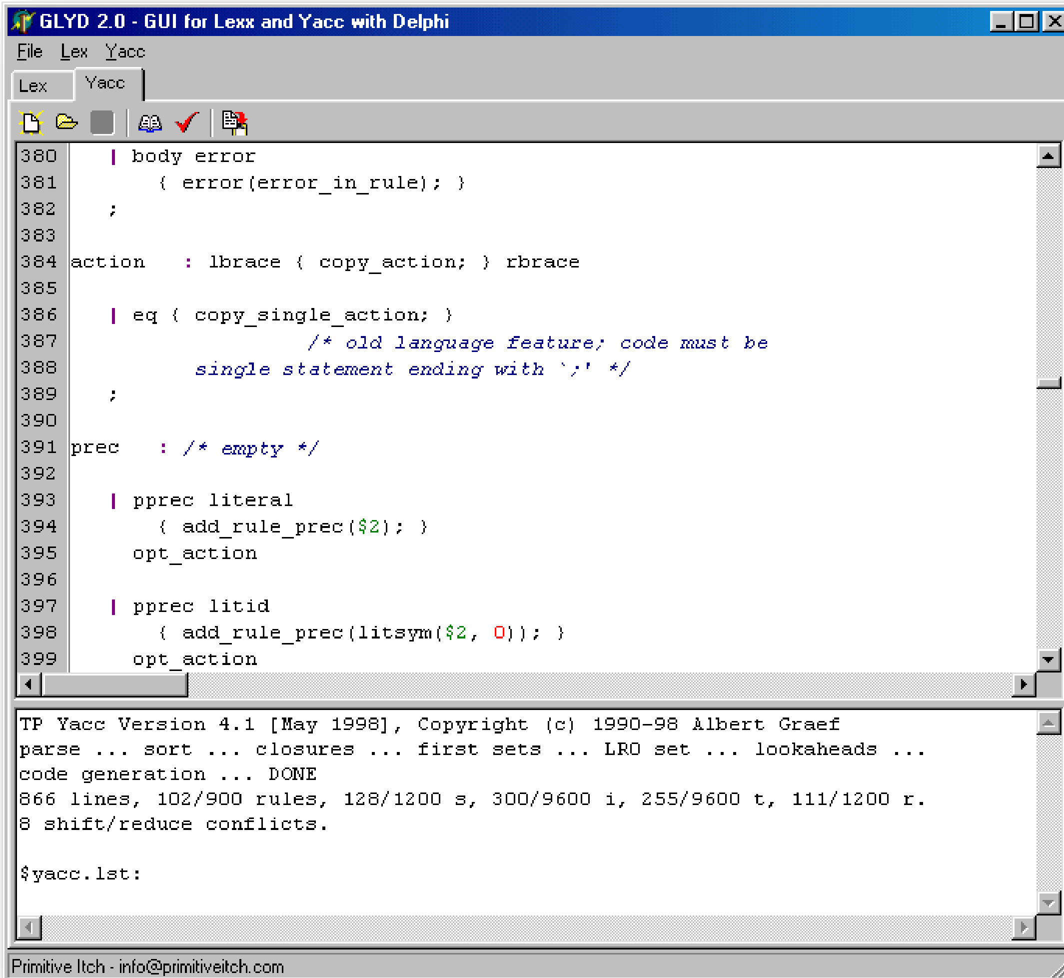
Stages of Compilation

- Source code
- Lexing
- Parsing
- IR
- Analysis
- Code generation
- Executable code



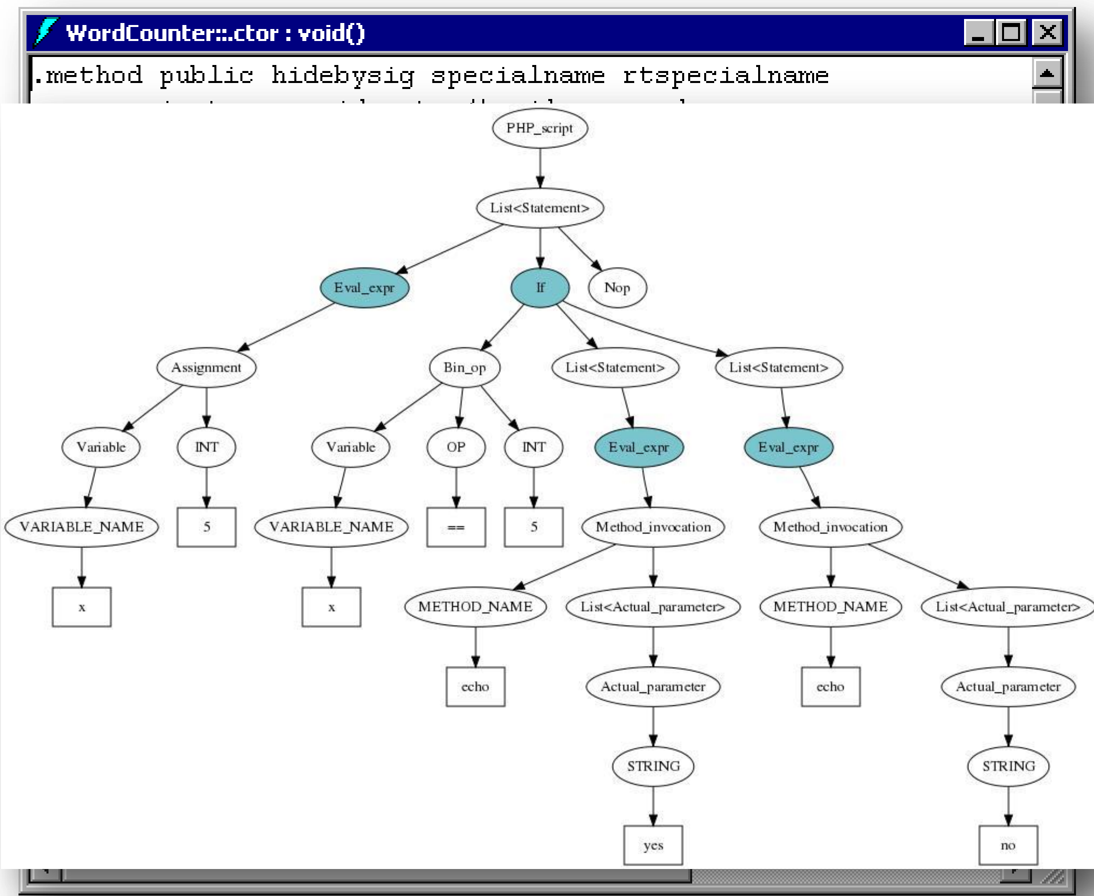
Stages of Compilation

- Source code
- Lexing
- Parsing
- IR
- Analysis
- Code generation
- Executable code



Stages of Compilation

- Source code
- Lexing
- Parsing
- IR
- Analysis
- Code generation
- Executable code



Stages of Compilation

- Source code
- Lexing
- Parsing
- IR
- Analysis
- Code generation
- Executable code

Stages of Compilation

Source code

Lexing

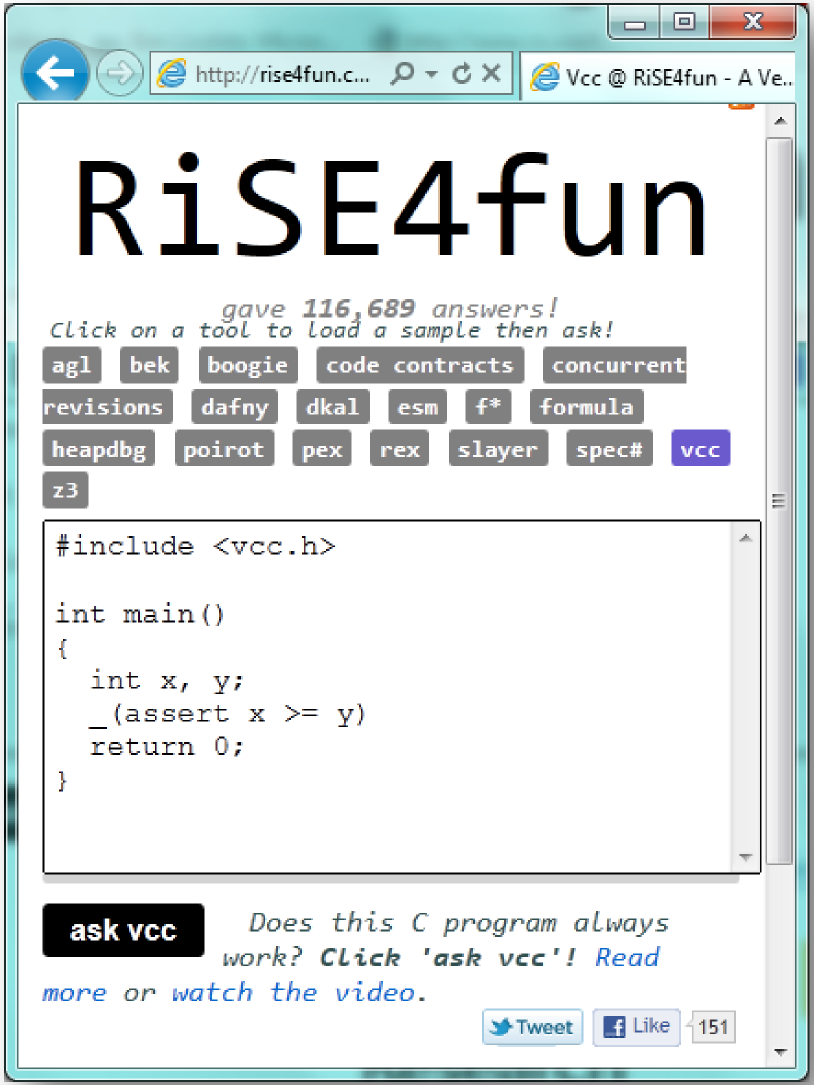
Parsing

IR

Analysis

Code generation

Executable code



Static Analysis

- Pros?

- Cons?

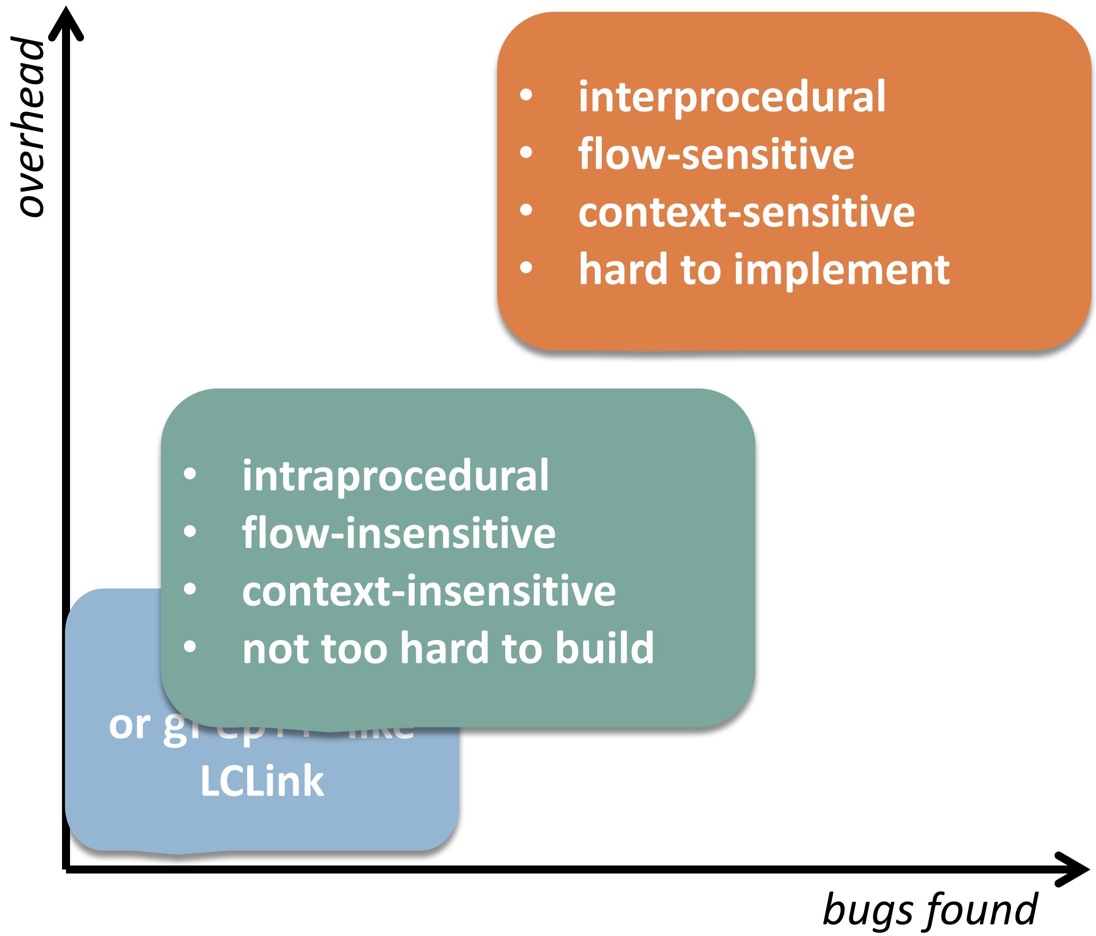
Static Analysis Tool for Bug Finding: Plan

1. Read the program
2. Transform into an Intermediate Representation (IR)
3. Do analysis on the IR
4. Output results

Dimensions of Analysis

- Intraprocedural vs. interprocedural
- Flow sensitive vs. flow-insensitive
- Context sensitive vs. context-insensitive

Cost vs. Effectiveness




SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:775–802

SP&E

A static analyzer for finding dynamic programming errors

William R. Bush*, Jonathan D. Pincus and David J. Sielaff

Intrinsa Corporation, Mountain View, CA, U.S.A.

SUMMARY

There are important classes of programming errors that are hard to diagnose, both manually and automatically, because they involve a program's dynamic behavior. This article describes a compile-time analyzer that detects these dynamic errors in large, real-world programs. The analyzer traces execution paths through the source code, modeling memory and reporting inconsistencies. In addition to avoiding false paths through the program, this approach provides valuable contextual information to the programmer who needs to understand and repair the defects. Automatically-created models, abstracting the behavior of individual functions, allow inter-procedural defects to be detected efficiently. A product built on these techniques has been used effectively on several large commercial programs. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: program analysis; program error checking

INTRODUCTION

There are important classes of programming errors that are hard to diagnose, both manually and automatically, because they involve a program's dynamic behavior. They include invalid pointer references, faulty storage allocation, the use of uninitialized memory, and improper operations on resources such as files (trying to close a file that is already closed, for example).

Finding and fixing such errors is difficult and expensive. They are usually found late in the development process. Extensive testing is often needed to find them, because they are commonly caused by complex interactions between components. Our measurements indicate that in commercial C and C++ code, on the order of 90% of these errors are caused by the interaction of multiple functions. In addition, problems may be revealed only in error conditions or other unusual situations, which are difficult to provoke by standard testing methods.

Traditional checking provided by the error-checking portion of compilers identifies errors relating to the static expression of a program, such as syntax errors, type violations, and mismatches between

*Correspondence to: William R. Bush, 1739 Lexington Avenue, San Mateo, CA 94402-4024, U.S.A.

Copyright © 2000 John Wiley & Sons, Ltd.

Received 9 November 1998
Revised 27 December 1999
Accepted 28 December 1999

Historical background

- Intrinsa
  - 1997-200?
  - paved way for MS
- Coverity
  - Out of Stanford
  - Commercial static analysis tools
- Fortify
  - Tools for security
- Klockwork

Paper Contributions

- Interprocedural static analysis algorithm
  - Address dynamic language features
  - Hash table use
  - Regular expression matching

- Features
  - Symbolic execution inside basic blocks
  - Basic block summaries

Paper Contributions

- Focus
  - SQL injection vulnerabilities. Why? Good idea?
  - XSS – claim to handle with minor modifications
- Experiments
  - 6 PHP apps
  - Finds 105 previously unknown vulnerabilities

PHP Language Features

- Natural SQL integration
  - $rows = mysql_query(
    “UPDATE users SET pass=‘$pass’ WHERE userid=‘$userid’”);

- Dynamic types and implicit casts
  - If ($userid < 0) exit;
  - $query = “SELECT * from users
    WHERE userid=‘$userid’”;

- Global environment
  - $_GET[‘name’] or $name
  - $ used with register_globals = on? Attacker may provide arbitrary value for $superuser by inserting something like $superuser=1 into HTTP request

Analysis Steps (Section 3)

$$
\begin{array}{c}
\text{PHP Source} \\
\Big\downarrow \text{Standard PHP Parser} \\
\text{Abstract Syntax Trees} \\
\Big\downarrow \forall\ \text{Function} \\
\text{Control Flow Graph} \\
\Big\downarrow \text{Intrablock Analysis} \\
\text{Block Summary} \\
\Big\downarrow \text{Intraprocedural Analysis} \\
\text{Function Summary} \\
\Big\downarrow \text{Interprocedural Analysis} \\
\text{Result}
\end{array}
$$

Basic blocks: Simulation

- Build up a model mapping labels -> values
- Special treatment of strings. Why?
- Special treatment of (some) booleans. Why?

Various Data Types: Representation

Strings Most fundamental type

- Concatenation of string segments
- `contains`(σ): String with substrings from a set σ of memory locations

Basic Block Summary

Set	Symbol	Description
Error set	E	Input variables which must be sanitized before entering this basic block
Return value	R	Representation for return value
Untaint set	U	Sanitized locations for each successor
Termination predicate	T	Block contains exit() or calls another termination function
Value flow	F	Set of location pairs (l_1, l_2) where l_1 is a substring of l_2 on exit
Definitions	D	Defined memory locations

Function Summary

Set	Symbol	
Error set	E	Inpu…
Return value	R	Repre…
Sanitized values	S	Saniti…
Program exit	X	Block… termi…

Memory location that can flow to database inputs

for main function, this cannot include
$_GET[…] or $_POST[…]

Function Summary

Set	Symbol	
Error set	E	Input en
Return value	R	
Sanitized values	S	Saniti
Program exit	X	Block termi

string-typed parameters or globals that might be returned, either fully or as part of a longer string

```
function make query($user, $pass) {
  global $table;
  return "SELECT * from $table ".
    "where user = $user and
                  pass = $pass";
}
```

R = {$table, $arg#1, $arg#2}

Function Summary

Set	Symbol	
Error set	E	Input ent
Return value	R	
Sanitized values	S	Saniti
Program exit	X	Block termi

the set of parameters or global variables that are sanitized on function exit

```
function is_valid($x) {
  if (is numeric($x)) return true;
  return false;
}
```

S = (false => {}, true => {arg#1})

Function Summary

Set	Symbol	Description
Error set	E	Input variables which must be sanitized before entering this basic block
Return value	R	Representation for return value
Sanitized values	S	Saniti
Program exit	X	Block

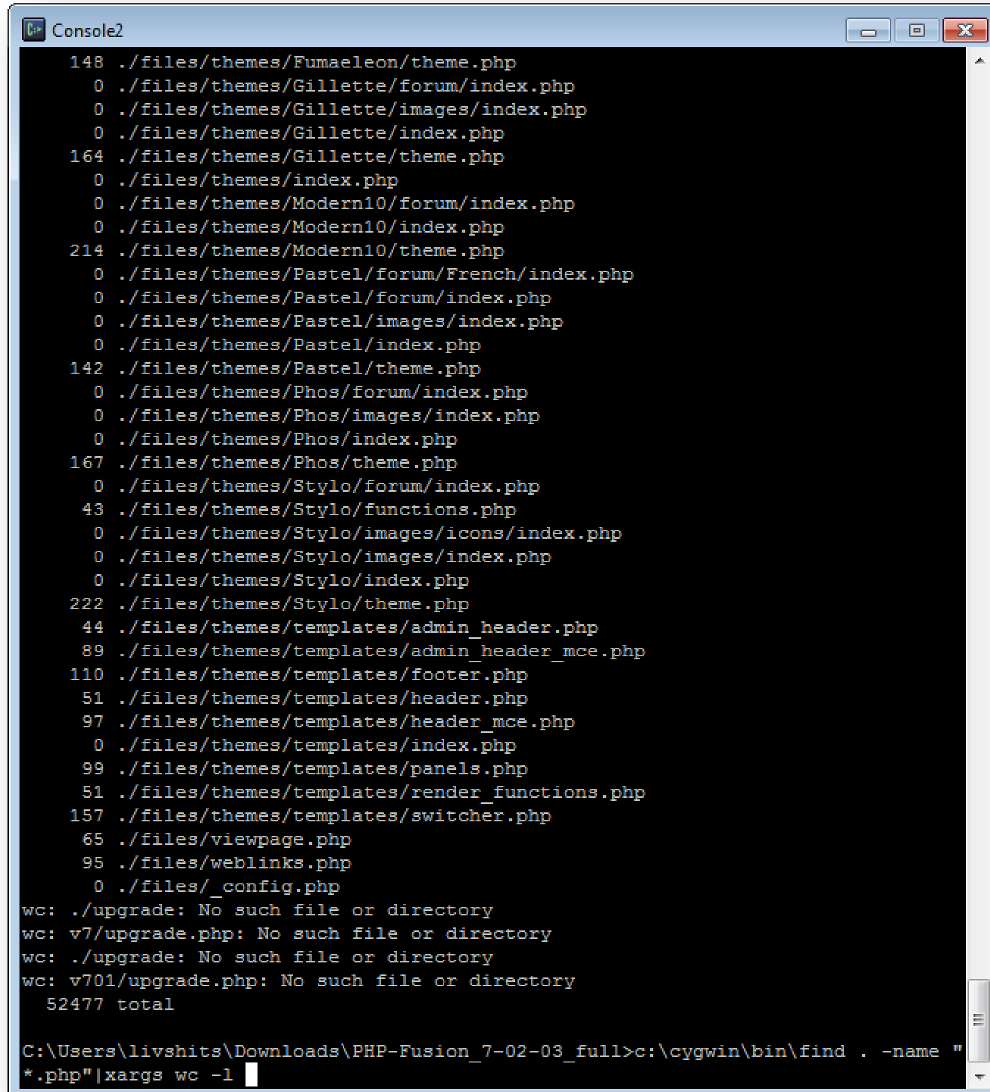
a Boolean which indicates whether the current function terminates program execution on all paths

Interprocedural Analysis

Since we require the summary information of a function before we can analyze its callers, the order in which functions are analyzed is important. Due to the dynamic nature of PHP (e.g., include statements), we analyze functions on demand—a function f is analyzed and summarized when we first encounter a call to f. The summary is then memoized to avoid redundant analysis. Recursive function calls are rare in PHP programs. If we encounter a cycle during the analysis, our current implementation uses a dummy "no-op" summary as a model for the second invocation.

Why On Demand?

- PHP Fusion
- version 7-02-03
- about 52K lines of code
- But really only about 16,000 matter



Checker Input

- We seed the checker with a small set of query functions (e.g. `mysql_query`) and sanitization operations (e.g. `is_numeric`).
- The checker infers the rest automatically

Checker Output

- Errors
  - Variables controlled by the attacker $_GET[…] and $_POST[…]
- Warnings
  - Other environment-define variables at the level of main

Result Summary

	Err Msgs	Bugs	(FP)	Warn
e107	16	16	(0)	23
News Pro	8	8	(0)	8
myBloggie	16	16	(0)	23
DCP Portal	39	39	(0)	55
PHP Webthings	20	20	(0)	6
Total	99	99	(0)	115

Table 1: Summary of experiments. Err Msgs: number of reported errors. Bugs: number of confirmed bugs from error reports. FP: number of false positives. Warn: number of unique warning messages for variables of unresolved origin (uninspected).

question of the day

Are the techniques in the paper sound, i.e. do they find all SQL injection bugs?

Runtime Analysis Overview

- Blacl … analysis
  - Fuz
  - Per … execution

Fuzzing: A Definition

"**Fuzz testing** or **fuzzing** is a software testing technique that provides invalid, unexpected, or random data to the inputs of a program. If the program fails (for example, by crashing or failing built-in code assertions), the defects can be noted."

Wikipedia

Why Fuzz in General?

- Another point of view of testing

- If its automated, why not?

- Some Fuzzing Successes:
  - Apple Wireless flaw DoS (MOKB-30-11-2006)
  - Month of Browser Bugs in 2006, many found with input fuzzing:
    - IE: 25
    - Safari: 2
    - Firefox: 2
    - Opera: 1
    - Konquerer: 1

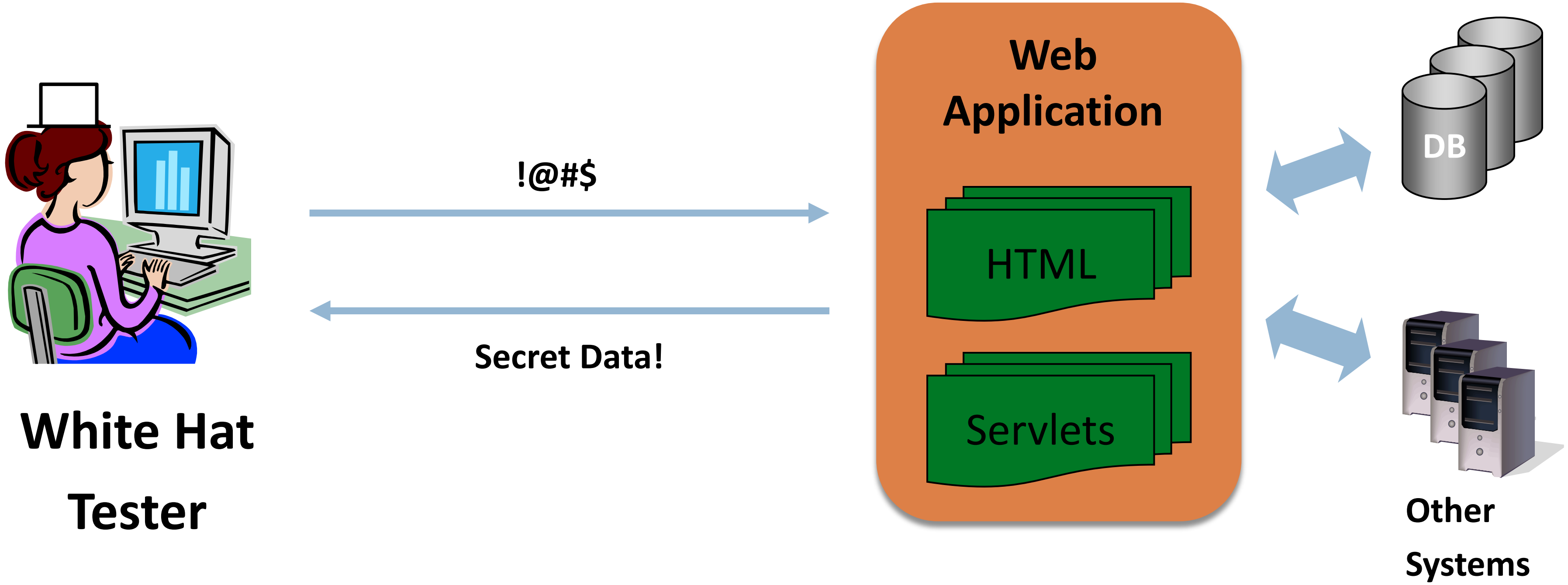
Need a Fuzzing Specification


```
setup "Webapp" do
  @host = "10.0.0.2"
  @port = 3000
  @headers = "HTTP_ACCEPT_CHARSET" => "utf-8,*"

  attack "search-box"
    many :get, "/sear
         :query => {:
    many :get, "/sear
         :query => {:
  end

  attack "post-page" do
    once :get, "/login.php
         {:user => :admi   :pass => :admin}
    many :post, "/post.php", :query =>
         {:title => word, :body => byte(50)}
  end
end
```

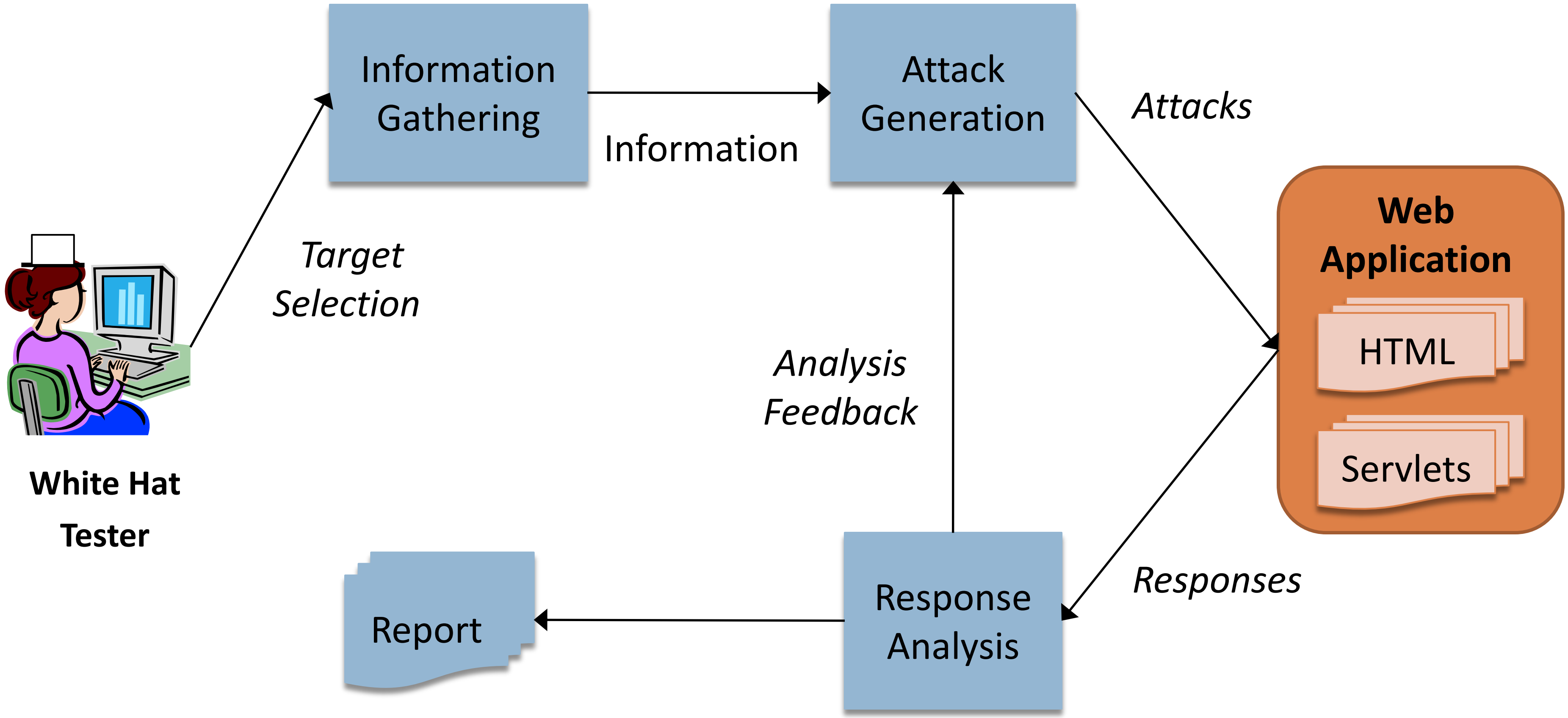
What do they look for?

Fuzz testing of web applications, Hammersland and Snekkenes

Penetration Testing Overview



Penetration Testing: Phases



Tainting

- Negative tainting
  - Mark *or taint* untrusted input data at runtime
  - Stop execution when untrusted input reaches “sinks”
- Positive tainting
  - Taint trusted data such as constant strings only
  - Stop execution when data reaching “sinks” is not tainted
- Propagate the taint through at the application executes

```
String s =
     req.getParameter(“userName”);
String s2 = “hello” + s;
output.println(“<div>”);
output.println(s2);
output.println(“</div>”);
```

Questions About Tainting

- How do we identify all sources in negative tainting?
- How do we remote taint?
- What is the runtime overhead?

Symbolic Execution

- Treat input values *symbolically*
- Propagate symbolic values through
- When encountering a conditional, consider both branches
- Use a *theorem prover* to eliminate infeasible paths

```
String s;
if (!P) {
  s = req.getParameter("userName");
} else {
  s = "";
}

String s2 = "hello" + s;
if (P) {
  output.println("<div>");
  output.println(s2);
  output.println("</div>");
} else {
  output.println("hello");
}
```

Summary

- Static analysis for bug finding
- Scripting languages analyzed (UsenixSec '05 paper)
- Runtime analysis
- Black-box
  - Fuzzing
  - Pen testing
- White-box
  - Tainting
  - Symbolic execution