Software is Full of Errors

- Error rate: 1–4.5 errors per 1000 lines
- Windows 2000
  - 35M LOC,
  - 63000 known bugs at time of release
  - 2 per 1000 lines
- Large consumer software
  - Formal specification & verification infeasible
Buffer Overruns

- A buffer access must stay within bounds
- Buffer overruns are responsible for over 50% of major vulnerabilities
- MS Blaster, Slammer, Code red, nimda, ..., Internet Worm, 1988
- Responsible for damages in billions
Memory Leaks

- All unused memory should be freed once and only once
- Memory exhaustion may cause long running programs to fail.
SQL Injection Errors

“Give me Bob’s credit card #”
or
“Delete all records”

- User may not supply SQL queries to databases directly
- One of top ten vulnerabilities
Happy-go-lucky SQL Query

User supplies: *name, password*

SELECT UserID, Creditcard
FROM Records
WHERE
  Name = ' + *name*
  + ' AND PW = ' + *password* + '
“—”: means “the rest are comments” in SQL

```
SELECT UserID, CreditCard FROM Records
WHERE:
Name = 'bob' AND PW = 'foo'
Name = 'bob'-- AND PW = 'x'
Name = 'bob' or 1=1-- AND PW = 'x'
Name = 'bob'; DROP Records-- AND PW = 'x'
```
Design Rules

- Buffer overruns, memory leaks, SQL injections
  - Same error is often repeated many times
- May be specific to a language, a class of applications, a program
- **Important**: may be critical to security
- **Succinct**: governs many lines of code
General Practice

- Design rules are implicit
- Violations of design rules rampant
- Tools: purify,
grep, emacs,
program environments
Leverage Computer Power

MOORE'S LAW

transistors

Intel® Itanium® Processor
Intel® Itanium® 2 Processor
Intel® Pentium® 4 Processor
Intel® Pentium® III Processor
Intel® Pentium® II Processor
Intel® Pentium® Processor
Intel486™ Processor
Intel386™ Processor
8086
80286
8080
8008
4004

1,000,000,000
100,000,000
10,000,000
1,000,000
10,000
1,000

Courtesy: Intel
Programs Don’t Grow Exponentially!

Size of Microsoft Windows

Million Lines of Code


NT3.1 95 98 NT5.0 XP 2000
This Talk

New generation of “Computer-Aided Programming Tools” to enforce critical software design rules
Custom Design Rule Checkers

- Intrinsa, Coverity: C checkers
- Built-in rules for operating systems
- Relatively simple analysis
- Unsound
- Found thousands of critical errors in Windows, Linux, BSD, …
New-Generation Tools

User-Supplied, Application-Specific

Automatic Extraction

Advanced Static Analysis

Dynamic Detection & Recovery

Design Rules
PQL: a Program Query Language
PQL: a Program Query Language

User-Supplied, Application-Specific

Advanced Static Analysis

Dynamic Detection & Recovery

Design Rules

Automatic Extraction
PQL Query

- Pattern:
  - Illegal sequences of operations on related objects
  - Looks like the simplest code excerpt with pattern

- Action:
  - Print out result, halt program, or recover
SQL Injection

PQL:

\[ x = \text{req.getParameter}(); \]
\[ \text{stmt.executeQuery}(x); \]

\[ \text{getParameter} \]
\[ x \]
\[ \text{executeQuery} \]

\[ o1 = \text{req.getParameter}(); \]
\[ \text{formal} = o1; \]
\[ o2.f = \text{formal}; \]
\[ o3.g = o2.f; \]
\[ \text{stmt.executeQuery}(o3.g); \]
Basic Question

\[ p_1 \text{ and } p_2 \text{ point to the same object?} \]
- Undecidable!
Dynamic Checker

$p_1$ and $p_2$ point to the same object?

- Instrument Java byte codes
- `getParameter`: record ID of $p_1$’s pointee
- `executeQuery`: check if $p_2$’s pointee has been recorded
Static Checker

$p_1$ and $p_2$ point to same object?

Pointer alias analysis
Pointer Alias Analysis

[Whaley & Lam, PLDI 2004]
**Pointer Analysis**

**Input Relations**
- $vPointsTo(v_1, h_1)$
- $vPointsTo(v_2, h_2)$
- $Store(v_1, f, v_2)$
- $Load(v_1, f, v_3)$

**Output Relations**
- $hPointsTo(h_1, f, h_2)$
- $vPointsTo(v_3, h_2)$

1. $v_1 = \text{new Object}();$
2. $v_2 = \text{new Object}();$
3. $v_1.f = v_2;$
4. $v_3 = v_1.f;$
Inference Rule in Datalog

Stores:

\[ h\text{PointsTo}(h_1, f, h_2) \leftarrow \text{Store}(v_1, f, v_2), \]
\[ \text{vPointsTo}(v_1, h_1), \]
\[ \text{vPointsTo}(v_2, h_2). \]

\[ v_1.f = v_2; \]
Pointer Alias Analysis

- Specified by 5 Datalog rules
  - Creation sites
  - Assignments
  - Stores
  - Loads
  - Type filter

- Apply rules until they converge
Method Invocations

- Context insensitive is imprecise
  - Unrealizable paths

```
Object id(Object x) {
    return x;
}
```

```plaintext
a = id(b);
c = id(d);
```
Context Sensitivity

- Context sensitivity is important for precision.
- Conceptually give each caller its own copy.

```java
Object id(Object x) {
    return x;
}

a = id(b);
c = id(d);
```
Cloning–Based Analysis

- Simple brute force technique.
  - Clone every path through the call graph.
  - Run context-insensitive algorithm on expanded call graph.
- The catch: exponential blowup
Cloning is exponential!
Recursion

- Actually, cloning is unbounded in the presence of recursive cycles.
- Technique: We treat all methods within a strongly-connected component as a single node.
Recursion
Top 20 Sourceforge Java Apps

Number of Clones

Number of clones

Size of program (variable nodes)

Number of Clones

10^16

10^12

10^8

10^4

10^0

1000 10000 100000 1000000
Cloning is infeasible (?)

- Typical large program has \(\sim 10^{14}\) paths
- If you need 1 byte to represent a clone:
  - 256 terabytes of storage
  - \(> 12\) times size of Library of Congress
  - 1GB DIMMs: $98.6 million
    - Power: 96.4 kilowatts (128 homes)
  - 300 GB hard disks: 939 x $250 = $234,750
    - Time to read sequential: 70.8 days
BDD comes to the rescue

- Many similarities across contexts.
  - Many copies of nearly-identical results.

- BDDs (binary decision diagrams) can represent large sets of redundant data efficiently.
Static Checker Generation

- **PQL Query**: 2 lines
- **Datalog**: 10 lines
- **BDD code**: 1000 lines
BDD: Binary Decision Diagrams
Call Graph Relation

- Call graph expressed as a relation.
  - Five edges:
    - Calls(A,B)
    - Calls(A,C)
    - Calls(A,D)
    - Calls(B,D)
    - Calls(C,D)
Call Graph Relation

- Relation expressed as a binary function.
  - A=00, B=01, C=10, D=11
Binary Decision Diagrams

- Graphical encoding of a truth table.

\[ \begin{array}{cccccccc}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]
Binary Decision Diagrams

- Collapse redundant nodes.
Binary Decision Diagrams

- Collapse redundant nodes.
Binary Decision Diagrams

- Collapse redundant nodes.
Binary Decision Diagrams

- Collapse redundant nodes.
Binary Decision Diagrams

- Eliminate unnecessary nodes.
Eliminate unnecessary nodes.
Binary Decision Diagrams

- Represent tiny and huge relations compactly
- Size depends on redundancy
  - Similar contexts have similar numberings
  - Variable ordering in the BDD
    - 10 minutes or “runs out of memory”
    - Active machine learning algorithm
Expanded Call Graph
Numbering Clones
Experience

- Context-sensitive points-to analysis
  - 800K byte codes in less than 20 minutes
- PQL suitable for many known error patterns
  - SQL injection
  - Resource leakage
  - Persistence object management
- Applied to 6 large programs unknown to us
- Found 44 critical errors easily
New-Generation Tools

User-Supplied, Application-Specific

Automatic Extraction

Advanced Static Analysis

Dynamic Detection & Recovery

Design Rules

PQL
Automatic Design Rule Extraction

- Too many design rules to specify
- Principle: Most of the code is correct
- Inconsistency = Errors
Clouseau: a static memory leak detector
Ownership Model

- **Owner pointer**
  - Obligated to delete object, or pass ownership to another pointer
- Every object has an owner pointer
- No memory leaks, no double deletes
Past Experience

- Decorate each function parameter with “own” “not-own”
- Not enforced → many mistakes
Assignment Statement

\[
\begin{align*}
&\text{a} = \text{new int;} \\
&\text{b} = \text{a;} \\
&\text{delete a;} \\
\end{align*}
\]

\[
\begin{align*}
&\text{a} = \text{new int;} \\
&\text{b} = \text{a;} \\
&\text{delete b;} \\
\end{align*}
\]

own(b), own(a)

\[
\begin{align*}
&\text{b} = \text{a} \\
&\text{own}(\text{b}) = 0 \\
&\text{own}(\text{a}) = \text{own}(\text{a}') + \text{own}(\text{b}')
\end{align*}
\]

own'(b), own'(a)
Clouseau

- Automatically infers function signatures
  - Based on “new” “delete”
  - Constraint: there is always 1 owning pointer
- Inconsistency: error
- A 125K-line commercial program
  - 50 lines of user specification on container structure (generic data types)
  - Root-causes 82% of memory leaked dynamically
  - Found many additional errors
- [Heine&Lam, PLDI 2003]
New-Generation Tools

- User-Supplied, Application-Specific
- Automatic Extraction
- Design Rules
- Advanced Static Analysis
- Dynamic Detection & Recovery
DIDUCE

Dynamic Invariant Deduction

∪

Checking Engine

(deduces, but sometimes incorrectly)
Motivation

- Difficulty in finding root cause of complex errors
- DIDUCE can potentially pinpoint the culprit automatically
Example

- For line # 1234,
  - Times 1–1000000: $0 \leq X \leq 3$
  - Time 1000001: $X = \text{0xa3d025ef}$

Aha! must be an error
DIDUCE Design

- Monitors every memory access!
- Learns signature for the data accessed by each bytecode from correct runs
- Report anomalies observed
- Anomalies signal cause of errors
Experience

- Pinpointed line of code causing errors
  - Errors in imap servers:
    Change triggers a latent error elsewhere
  - Memory subsystem simulator:
    otherwise unknown errors
- Reported corner cases of interest
Lessons

- Dumb but tireless
  - High overhead but pain-free
  - Unaffected by programmers’ misconception
- Finds many serendipitous design rules
- Need to trigger just 1 of the rules
- [Hangal & Lam, ICSE 02]
Summary

Design Rules

User-Supplied, Application-Specific

Automatic Extraction

Advanced Static Analysis

Dynamic Detection & Recovery
Conclusions

- New tools that exploit computing power to manage software complexity
- New advanced analyses
  - Pointer alias analysis—Binary decision diagrams
- Analysis available to programmers
  - PQL: easy to express execution patterns
- Automatic design rule extraction
  - Inconsistencies → errors