Carnegie Mellon
Software Engineering Institute

CERT
Analysis
Center

CERT Experience with Security Problems in Software

Tom Longstaff

June 2003

CERT® Centers
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

The CERT Coordination Center is part of the Software Engineering Institute. The Software Engineering Institute is sponsored by the U.S. Department of Defense.

Survivability

Survivability is the ability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents.

Survivability focus is on the system mission

- assume imperfect defenses and component failure
- analyze mission risks and tradeoffs
- identify decision points with survivability impact
- provide recommendations with business justification

CERT's Areas of Expertise

Vulnerability analysis

Artifact analysis

Insider threats

Survivable Architectures

Function abstraction/extraction

Modeling and simulation

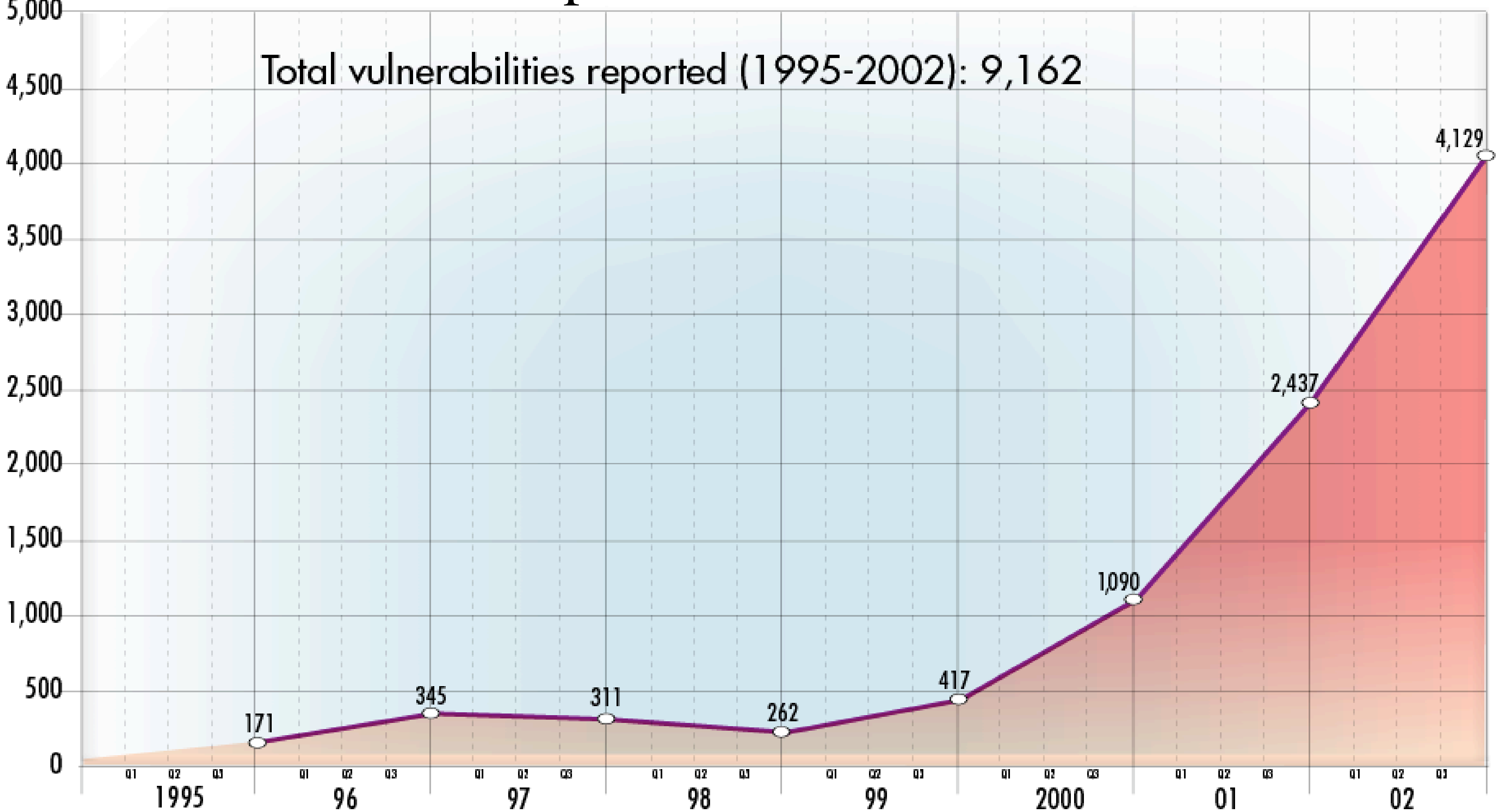
Dependency and critical infrastructure analysis

Best practices and methodologies for testing software

R&D

Critical Need for Better Software

Vulnerabilities Reported to CERT/CC



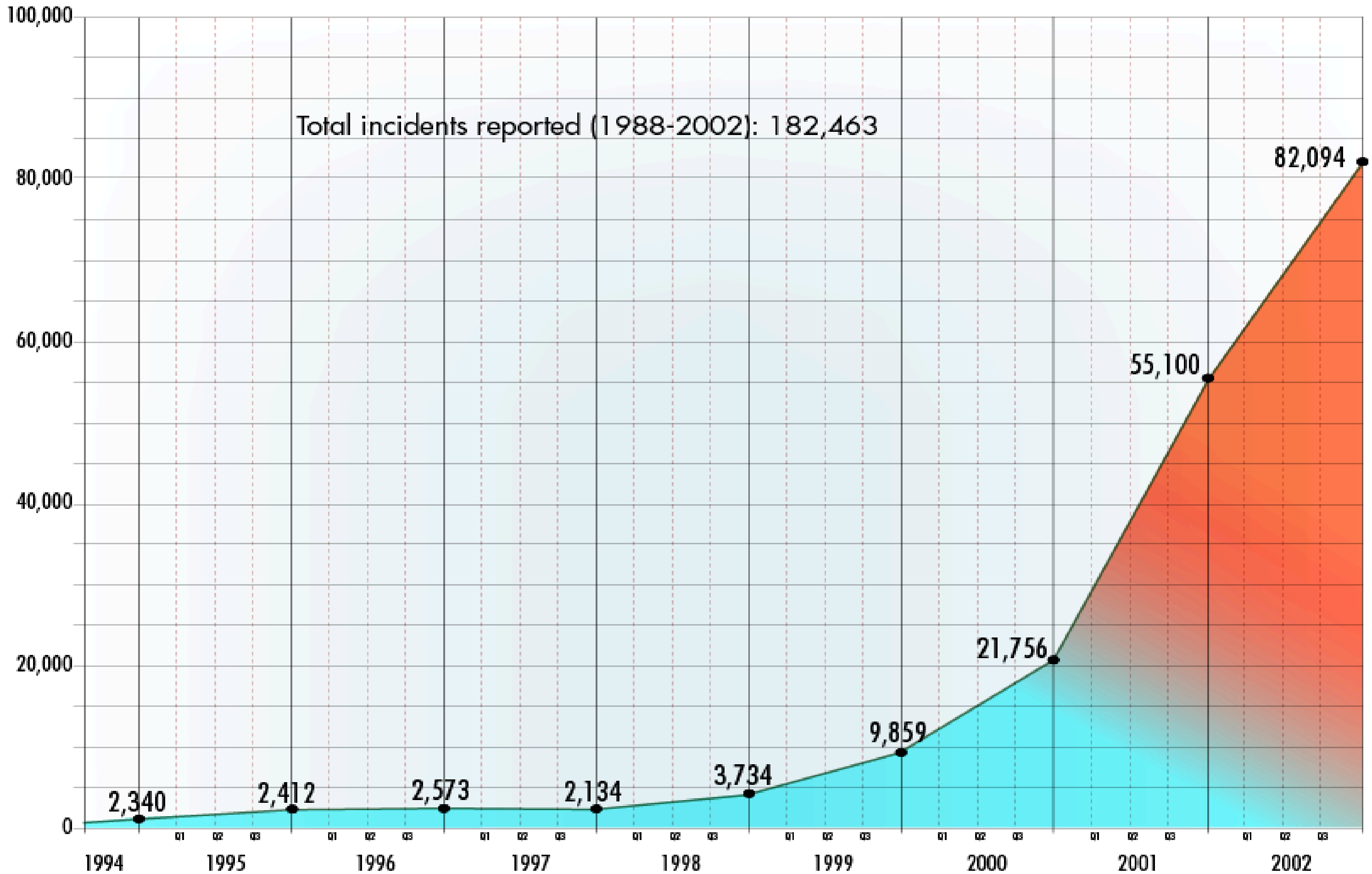
Rough Stats for 2003

Public Stats available on www.kb.cert.org

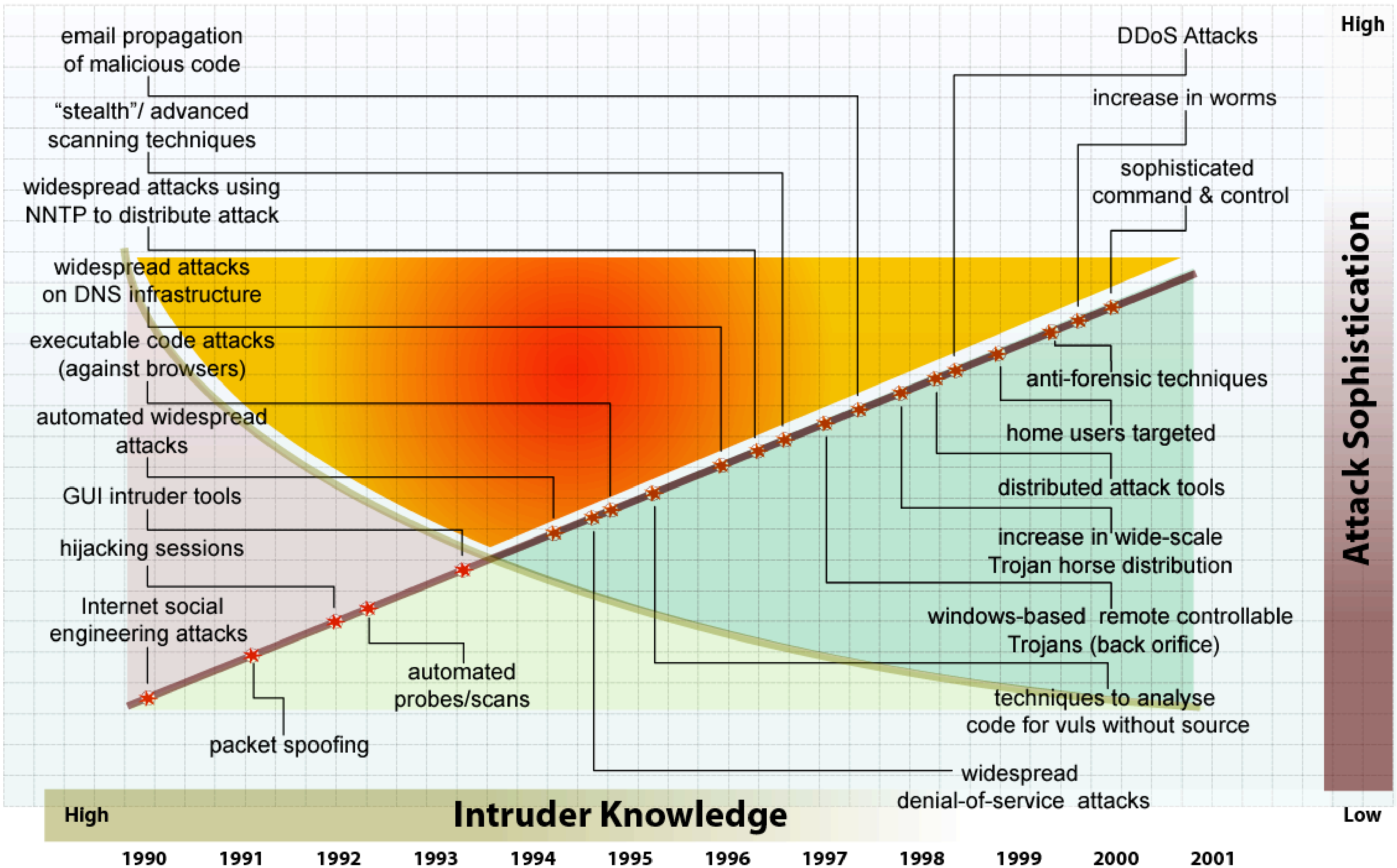
Buffer overflows	7
DoS	8
Java	3
Sendmail	2
Linux	12
Microsoft	22
Not Microsoft	80

Critical Need for Better Practices

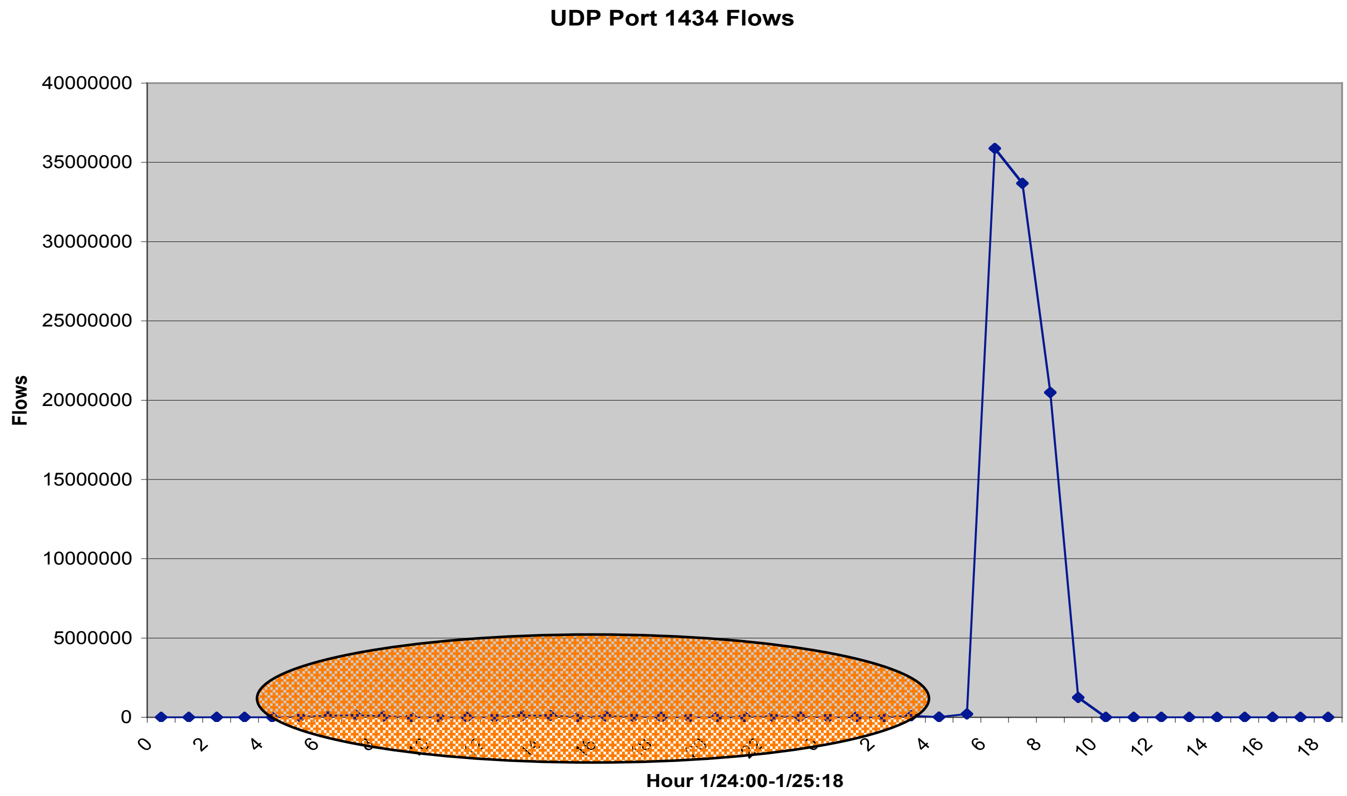
Incidents Reported to the CERT/CC



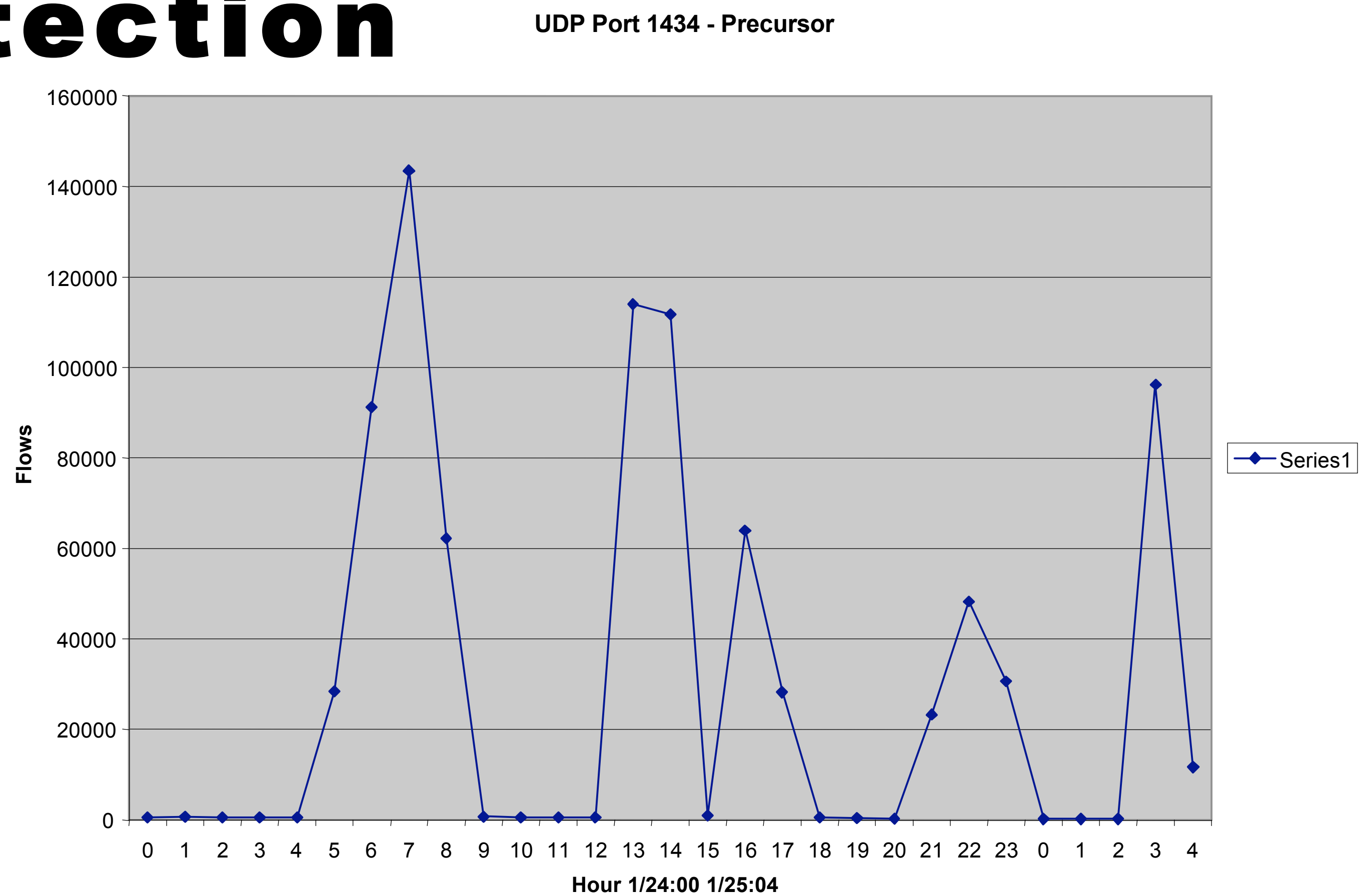
Incident Trends



Inbound Slammer Traffic



Slammer: Precursor Detection



Slammer: Precursor Analysis

Focused on hours 6, 7, 8, 13, 14

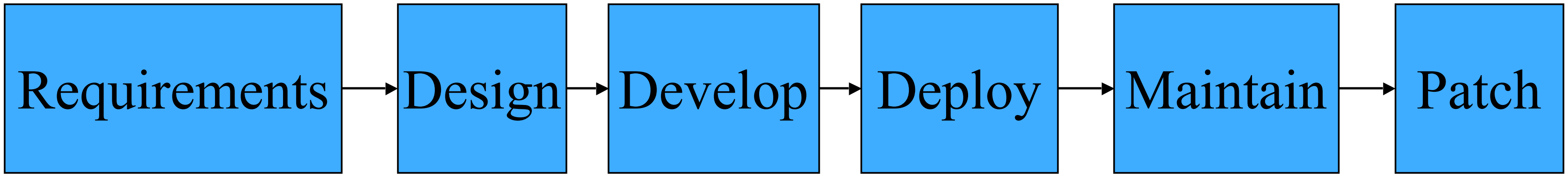
Identified 3 primary sources, all from a known adversary

All 3 used a fixed pattern

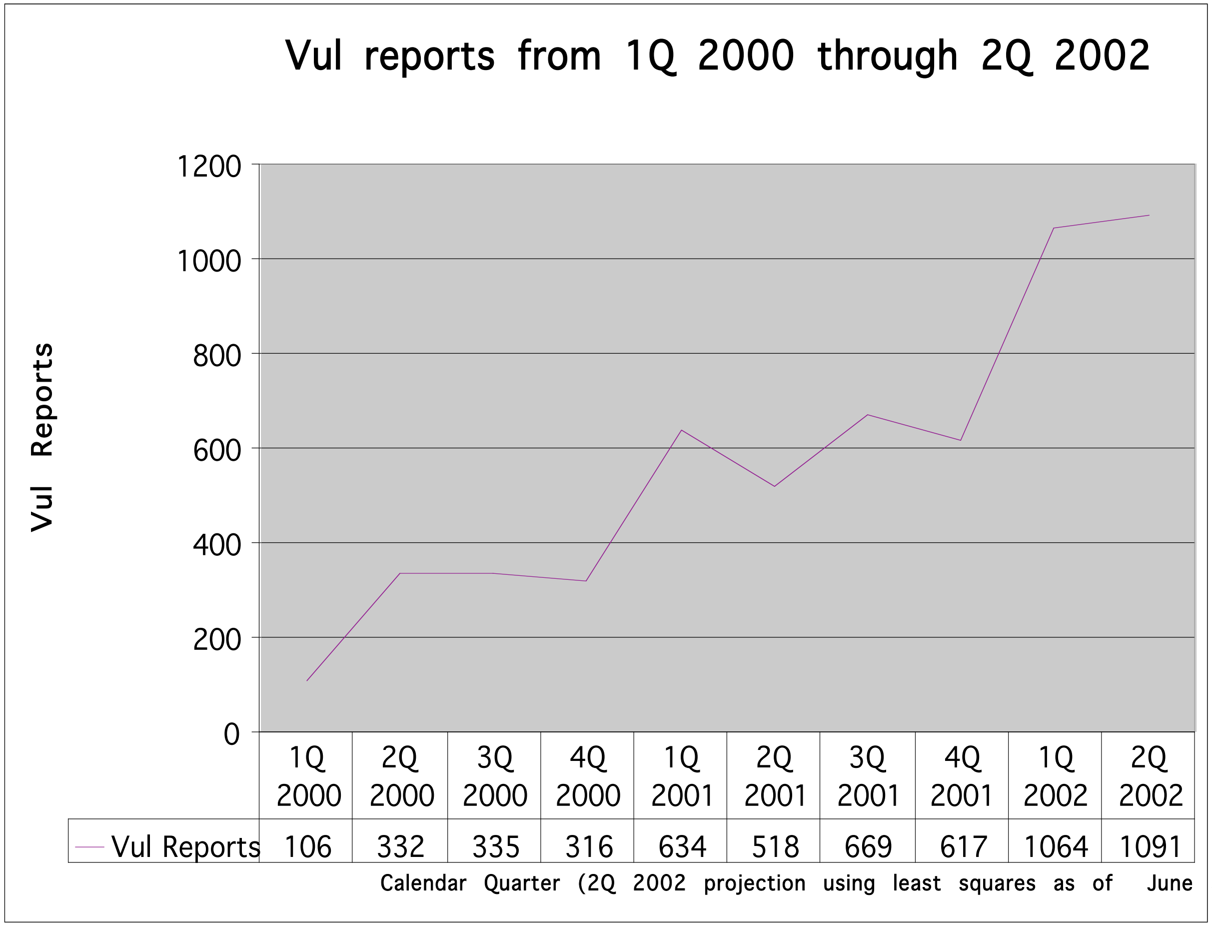
Identified responders: 2 out of 4 subsequently compromised

The Response Strategy

- The development of software for secure applications is handled the same way as other software.
- This typically results in many delivered defects, including security vulnerabilities (vuls).



There Are Many Vuls to Patch



	1Q 2000	2Q 2000	3Q 2000	4Q 2000	1Q 2001	2Q 2001	3Q 2001	4Q 2001	1Q 2002	2Q 2002
Vul Reports	106	332	335	316	634	518	669	617	1064	1091

The Administrative Workload

With 5500 vulnerabilities reported in 2002

- Somebody must read each vul description
  - 5500 * 20 minutes to read = 229 days
- If an organization is affected by 10% of the vuls
  - 550 vuls * 1 hour to install the patch = 69 days
- Just to read security news and patch a single system 229 + 69 = 298 days

With 5 minutes to read new bulletins and a 1% “hit rate”

- Just reading bulletins takes almost 65 days.
- This is over 25% of an administrators time.

What is a Vulnerability?

Different people have different definitions. The CERT/CC has an internal understanding that a vulnerability:

- Violates an explicit or implicit security policy
- Is usually caused by a software defect
- That similar defects are the same vulnerability (e.g. SNMP was 2 vulnerabilities)
- Often causes unexpected behavior

We specifically exclude from "vulnerability":

- Trojan horse programs (evil email attachments)
- Viruses and Worms (self propagating code)
- Intruder tools (scanners, rootkits, etc.)

Vulnerabilities are the technical problems that permit these things to exist

The Homerun Vul vs. 3 singles and a Double

Most of the most widely discussed vulnerabilities are "homerun" vuls -- they get you as much as you can get all in one fell swoop

But three singles and a double will probably score 2 or 3 runs

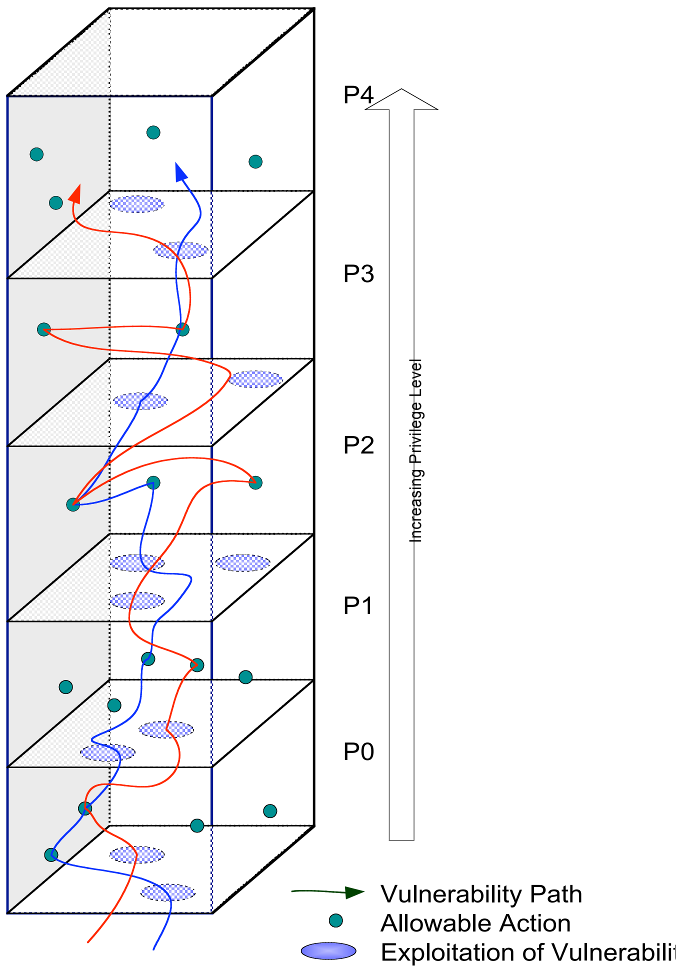
We use a reasoning system to see how "singles" can be used to do real damage

Vul Chaining: Project Goal

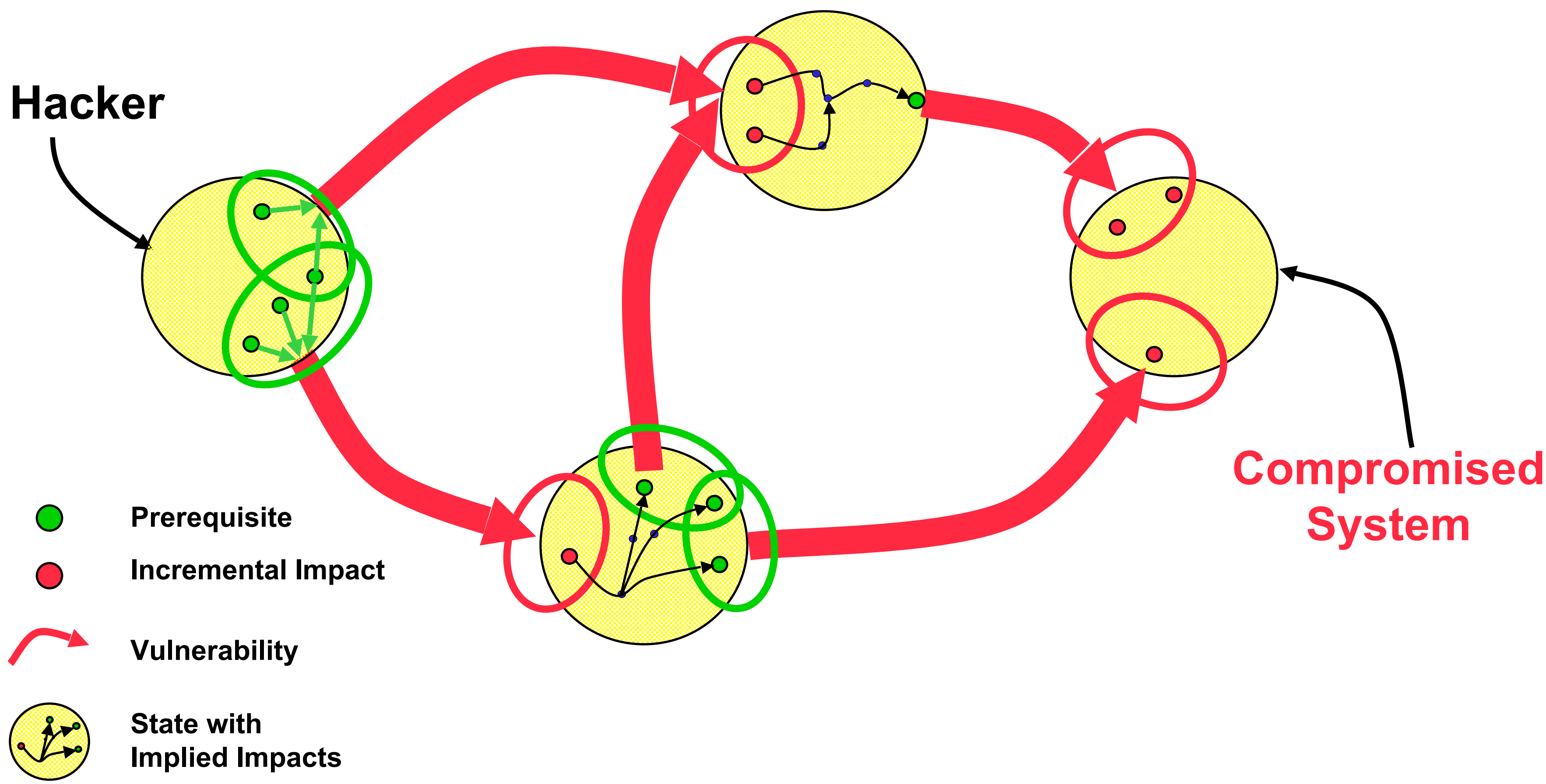
Model the paths that could be used by attackers when compromising a system

- Privilege Level
- Incremental Impact
- Implied Impact

Use the model to better understand how a system is compromised



Vulnerability Graphs



Proof-of-Concept

Model Microsoft Windows and Some Applications

- Complex privilege system
  - Multiple levels
- Ample sample space

Model Windows “vulnerabilities”

- Good documentation on the nature of the vulnerabilities

Place the model into an automated reasoning system

Vul Chaining Status

Performed basic research to understand vulnerabilities and how they are related

- More rigorous description
- Functional relationship

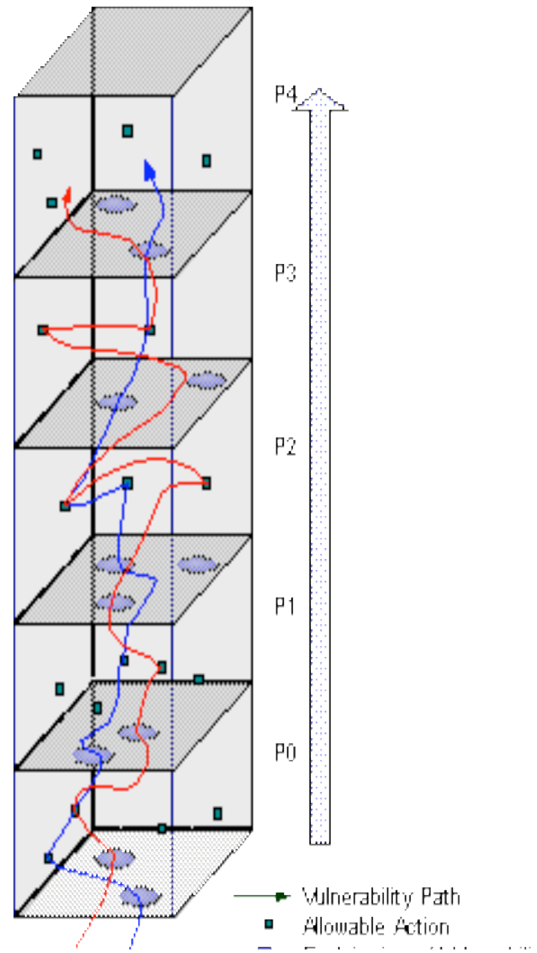
Developed a prototype system

- Sample vulnerability graphs

Designed complete system

Results to Be Published in Bell Labs Technical Journal

Seeking Additional Funding to Continue this Work



Critical Need for Better Engineering Methods

Sophisticated intruders target

- distributed user workflows
- trust relationships among distributed systems
- limited visibility into and control of remote systems
- people and the meaning they assign to content
- work resources that people rely on

Many organizations rely solely on insufficient boundary control and “bolt-on” mechanisms as defense

Resistance, recognition, and response must be integrated into the system and application architecture

21st Century State of Practice

- Society depends on systems whose full behavior is not known
- No programmer can say for sure what a sizable program does in all uses
- Planted vulnerabilities and malicious behavior cannot be detected reliably in delivered software

Vulnerable to More Than Simple Accidents

- "Trustworthy" software developers?
- Offshore software production
- Planted behavior in delivery channel
- Reused code from unknown sources

It is hard enough to find accidental vulnerabilities

Deliberately hidden malicious code is beyond today's capability to detect

Why is this such a hard problem?

Software life cycle is complex, with many opportunities to introduce malicious behavior

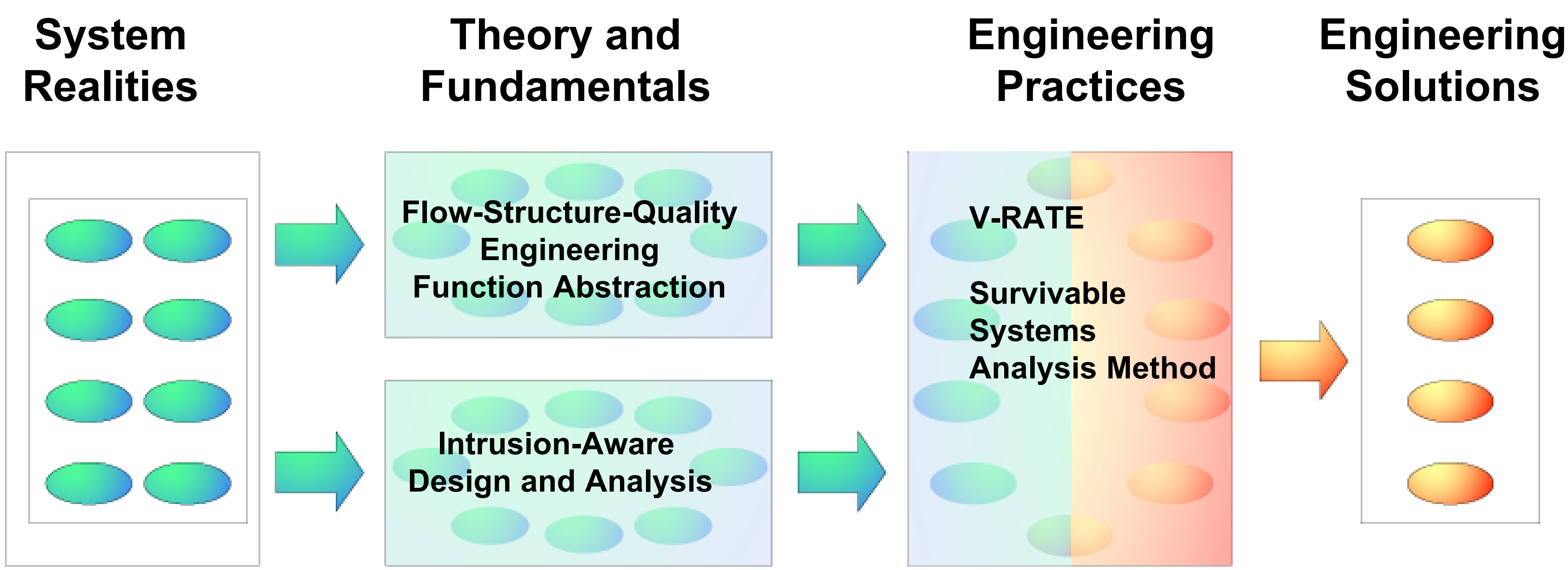
Testing focuses on direct functionality and failures, not at uncovering hidden behavior

Frequently hard to distinguish between accidental vulnerabilities and planted malicious code

Business makes use of “hidden” behavior to support intellectual property control

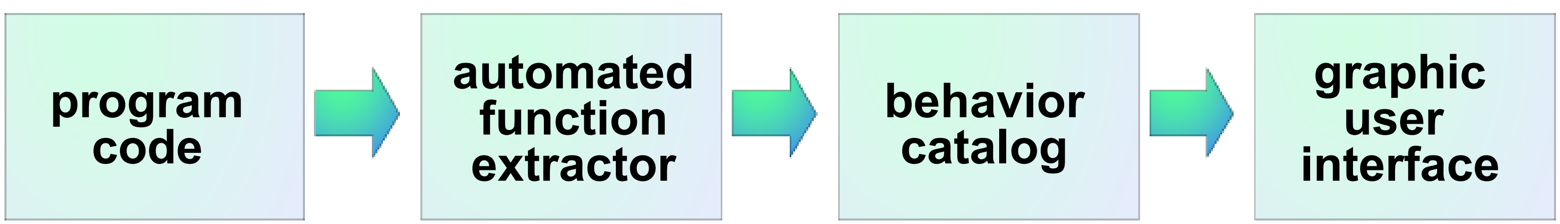
Survivability Engineering

Overall Objective: **Develop rigorous system engineering practices for mission survivability**



Function Extraction (FX) Project

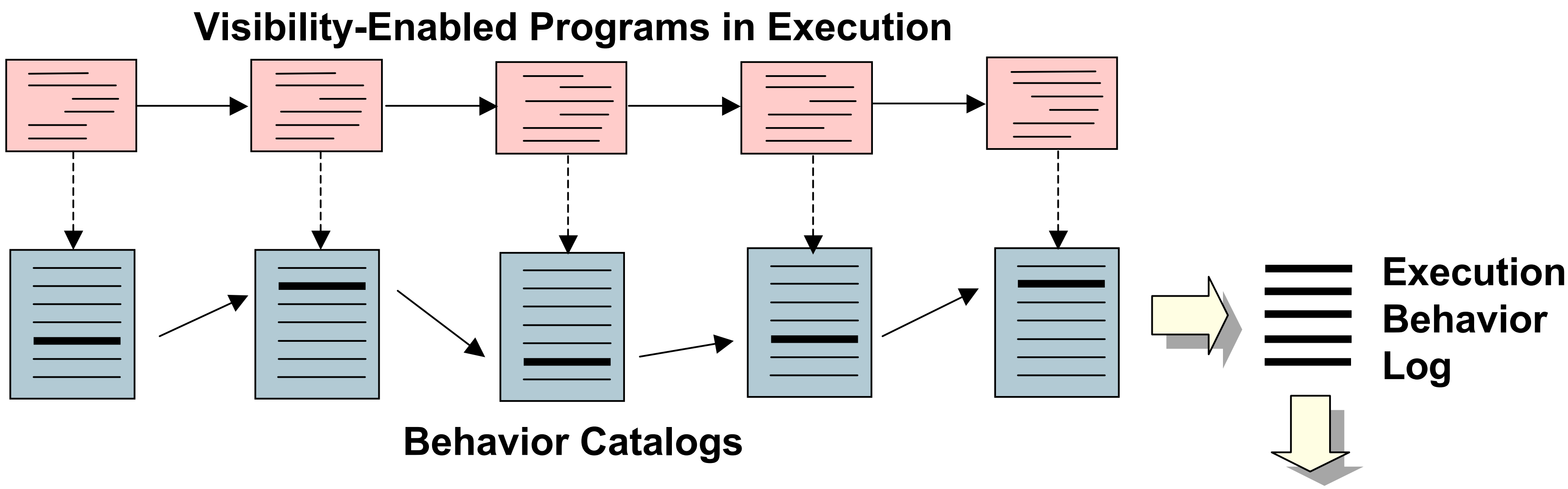
- Programmers lack means to say for sure what the behavior of sizable programs is in all uses.
- Unknown behavior is the source of many problems in software engineering.
- FX project goal is automated calculation of the full functional behavior of programs.
  - Transform behavior discovery from error-prone process in human time scale to precise process in CPU time scale
  - Potentially transformational technology for software and security engineering

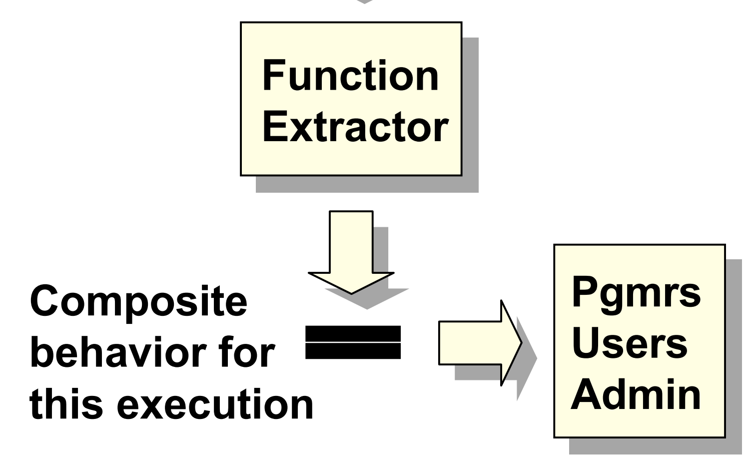


Function Extraction Technology

- Programs and their parts are implementations of mathematical functions or relations (mappings from domains to ranges)
- These functions can be extracted by stepwise abstraction with mathematical precision in an algebra of functions
- Development of automated extractors is difficult (total solution theoretically impossible) but feasible

Dynamic Use of Extractors





Use explanation of behavior to:

- Understand composition
- Examine correctness
- Simplify architecture
- Impose control

Software Analysis

```
public class AccountRecord {
  public int acct_num;
  public double balance;
  public int loan_out;
  public int loan_max;
} // end of AccountRecord

public class AdjustRecord
extends AccountRecord {
  public bool default;
} // end of AdjustRecord

public static AdjustRecord classify_account
(AccountRecord acctRec) {
  AdjustRecord adjustRec = new AdjustRecord();
  adjustRec.acct_num = acctRec.acct_num;
  adjustRec.balance = acctRec.balance;
  adjustRec.loan_out = acctRec.loan_out;
  adjustRec.loan_max = acctRec.loan_max;
  adjustRec.default = (adjRec.balance < 0.00);

  while ((adjustRec.balance < 0.00) &&
      (adjustRec.loan_out + 100) <= adjustRec.loan_max))
  {
    adjustRec.loan_out = adjustRec.loan_out + 100;
    adjustRec.balance = adjustRec.balance + 100.00;
  }

  return adjustRec;
}
```

What Does This Code Do?

- Unlike other engineering disciplines, software engineering has no practical means to fully evaluate the expressions it produces

Software Analysis Today

```
public class AccountRecord {
   public int acct_num;
   public double balance;
   public int loan_out;
   public int loan_max;
} // end of AccountRecord

public class AdjustRecord
extends AccountRecord {
   public bool default;
} // end of AdjustRecord

public static AdjustRecord classify_account
(AccountRecord acctRec) {
   AdjustRecord adjustRec = new AdjustRecord();
   adjustRec.acct_num = acctRec.acct_num;
   adjustRec.balance = acctRec.balance;
   adjustRec.loan_out = acctRec.loan_out;
   adjustRec.loan_max = acctRec.loan_max;
   adjustRec.default = (adjRec.balance < 0.00);

   while ((adjustRec.balance < 0.00) &&
         (adjustRec.loan_out + 100) <= adjustRec.loan_max))
   {
      adjustRec.loan_out = adjustRec.loan_out + 100;
      adjustRec.balance = adjustRec.balance + 100.00;
   }

   return adjustRec;
}
```

- Has been a problem for 40 years
- Read code to learn function, find malicious properties
- Hard, haphazard, error-prone
- Human time scale, fallibilities
- Laborious process produces suspect knowledge
- Change a line and start over
- But visibility is vital to detecting malicious code

Software Analysis Tomorrow

Program

```
public class AccountRecord {
   public int acct_num;
   public double balance;
   public int loan_out;
   public int loan_max;
} // end of AccountRecord

public class AdjustRecord
extends AccountRecord {
   public bool default;
} // end of AdjustRecord

public static AdjustRecord classify_account
(AccountRecord acctRec) {
   AdjustRecord adjustRec = new AdjustRecord();
   adjustRec.acct_num = acctRec.acct_num;
   adjustRec.balance = acctRec.balance;
   adjustRec.loan_out = acctRec.loan_out;
   adjustRec.loan_max = acctRec.loan_max;
   adjustRec.default = (adjRec.balance < 0.00);

   while ((adjustRec.balance < 0.00) &&
         (adjustRec.loan_out + 100) <= adjustRec.loan_max))
   {
      adjustRec.loan_out = adjustRec.loan_out + 100;
      adjustRec.balance = adjustRec.balance + 100.00;
   }
   return adjustRec;
}
```

Automated Behavior Extraction

Business rules (design spec) in three cases

Behavior Catalog

1. AccountRecord acctRec
   Object is unchanged
2. AdjustRecord adjustRec
   A new object adjustRec is created and returned, the contents of which are described in three cases:

```
CASE 1:
if (acctRec.balance >= 0.00)
then
   adjustRec.acct_num = acctRec.acct_num
   adjustRec.balance   = acctRec.balance
   adjustRec.loan_out  = acctRec.loan_out
   adjustRec.loan_max = acctRec.loan_max
   adjustRec.default    = false
CASE 2:
if (acctRec.balance < 0.00) and
   (acctRec.loan_out + 100> acctRec.loan_max)
then
   adjustRec.acct_num = acctRec.acct_num
   adjustRec.balance   = acctRec.balance
   adjustRec.loan_out  = acctRec.loan_out
   adjustRec.loan_max = acctRec.loan_max
   adjustRec.default    = true
CASE 3:
if (acctRec.balance < 0.00) and
   (acctRec.loan_out + 100 <= acctRec.loan_max)
then
   adjustRec.acct_num = acctRec.acct_num
   adjustRec.balance   = acctRec.balance + (100.00 * term)
   adjustRec.loan_out  = acctRec.loan_out + (100 * term)
   adjustRec.loan_max = acctRec.loan_max
   adjustRec.default    = true
where
   term   = min(term1, term2)
   term1 = ceiling(0.00 – acctRec.balance)/100.00)
   term2 = 1 + floor((acctRec.loan_max – 100 –
   acctRec.loan_out)/100)
```

Detecting Malicious Code

```
  public int loan_out;
  public int loan_max;
} // end of AccountRecord

public class AdjustRecord
extends AccountRecord {
  public boolean in_default;
  public static AdjustRecord slush;
} // end of AdjustRecord

public static AdjustRecord classify_account
(AccountRecord acctRec) {
  AdjustRecord adjustRec = new AdjustRecord();
  adjustRec.acct_num = acctRec.acct_num;
  adjustRec.balance = acctRec.balance;
  adjustRec.loan_out = acctRec.loan_out;
  adjustRec.loan_max = acctRec.loan_max;
  adjustRec.in_default = (adjustRec.balance < 0.00);
  while ((adjustRec.balance < 0.00) &&
      ((adjustRec.loan_out + 100) <= adjustRec.loan_max)) {
    adjustRec.loan_out += 100;
    adjustRec.balance += 100.00;
  }
  if (adjustRec.balance < 0.00) {
    adjustRec.balance -= 0.01;
    AdjustRecord.slush.balance += 0.01;
  }
  return adjustRec;
}
```

Automated Behavior Extraction

```
1.  AccountRecord acctRec
              Object is unchanged
2.  AdjustRecord adjustRec
              A new object adjustRec is created and returned,
              the contents of which are described in four
CASE 1:
if (acctRec.balance >= 0.00)
   adjustRec.loan_max   = acctRec.loan_max
   adjustRec.in_default = false
CASE 2:
if (acctRec.balance < 0.00) and
  (acctRec.loan_out + 100 > acctRec.loan_max)
then
   adjustRec.acct_num   = acctRec.acct_num
   adjustRec.balance    = acctRec.balance
   adjustRec.loan_out   = acctRec.loan_out
   adjustRec.loan_max   = acctRec.loan_max
   adjustRec.in_default = true
CASE 3:
if (acctRec.balance < 0.00) and
  (acctRec.loan_out + 100 <= acctRec.loan_max) and
  (term1 <= term2)
then
   adjustRec.acct_num   = acctRec.acct_num
   adjustRec.balance    = acctRec.balance + (100.00 * term1)
   adjustRec.loan_out   = acctRec.loan_out + (100 * term1)
   adjustRec.loan_max   = acctRec.loan_max
   adjustRec.in_default = true
CASE 4:
if (acctRec.balance < 0.00) and
  (acctRec.loan_out + 100 <= acctRec.loan_max) and
  (term1 > term2)
then
   adjustRec.acct_num   = acctRec.acct_num
   adjustRec.balance    = acctRec.balance + (100.00 * term2) - 0.01
   adjustRec.loan_out   = acctRec.loan_out + (100 * term2)
   adjustRec.loan_max   = acctRec.loan_max
   adjustRec.in_default = true
   AdjustRecord.slush.balance = AdjustRecord.slush.balance + 0.01
where
   term1 = ceiling(0.00 – acctRec.balance)/100.00)
   term2 = 1 + floor((acctRec.loan_max – 100 –
          acctRec.loan_out)/100)
```

Malicious code case skims accounts

FX Bottom Line

- FX is a foundation for a new science of visible computing
- Opportunity to move software engineering into the visible computing era
- Modest investment now can produce substantial payoff

Future State

For Acquisition:

A community exists that can repeatedly examine code for undesired behavior

Methodologies and technologies exist to make this feasible and cost effective

For Development:

Software is more self-aware of its behavior and may determine when undesired behavior is attempted

The software engineering process supports full accountability of all members of the life cycle for responsibility

Emergent Algorithms

Survivability is an *emergent property* of a system.

Desired system-wide properties “emerge” from local actions and distributed cooperation.

An emergent property need not be a property of any individual node or link.

Collective or crowd behavior emerges from the rules for individuals and their interactions with their neighbors.

CERT and TSP

PSP-data shows that programmers inject a defect about every 10 lines of code written.

Most commercial applications have a defect density of about 2 defects per KSLOC (MS Win 2000, with 30 million LOC, was released with 63,000 known defects[1])

If only 5% of these defects were potential security concerns, there would be 100 security defects per MSLOC.

[1] Business Week On Line – Software Hell, Dec 1999 and
CNN Interactive – Will Bugs Scare Off Users Of Windows 2000, Feb 17, 2000

TSP and Secure Systems

The TSP provides a framework, a set of processes, and disciplined methods for producing quality software.

Software produced with TSP has one or two orders of magnitude fewer defects than current practice.

- 0.02 defects/KSLOC vs. 2 defects/KSLOC
- 20 defects per MSLOC vs. 2000 defects per MSLOC

If 5% of the defects are potential security holes, with TSP there would be 1 vulnerability per MSLOC.

TSP and Secure Systems

TSP also addresses the need for

- professional behavior
- a supportive environment
- sound software engineering practice
- operational processes
- software metrics

TSP could be extended to provide the process, training, and support required to consistently produce secure software products.

TSP For Secure Systems

TSP for Secure Systems is a joint effort of the TSP team and SEI's NSS (CERT) group.

The work is based on proven TSP quality practices and CERT's extensive security skills and knowledge.

TSP secure augments PSP training and TSP introduction with specialized security training.

- secure design process
- secure implementation practices
- secure review and inspection methods
- secure test process
- security-related predictive measures

TSP For Secure Systems

The goal of the project is to develop a TSP-based method that can predictably produce secure software.

The TSP for Secure Systems project is developing a process and support system that will

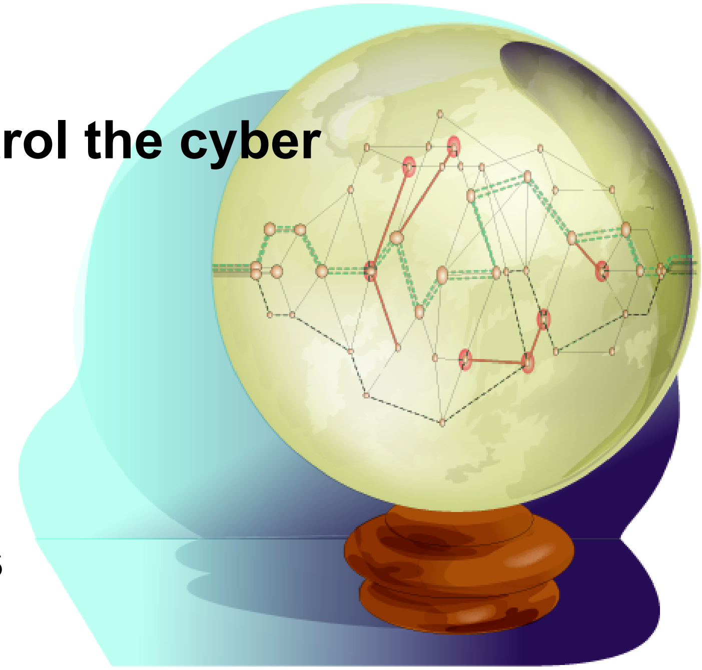
- support secure systems development practices
- predict the likelihood of latent security defects
- be dynamically tailored to respond to new threats

TSP for Secure Systems will be tested in several pilots.

Some Questions

- Can development practices that lead to security defects be identified?
- Can a process be developed to implement these practices?
- Can measures and tools be developed to establish predictability?
- What are the design principles for secure software?
- Can “security patterns” be defined?
- Can the vulnerability of software be quantified?
- Can repair costs be predicted?
- What are the properties of security defects?
  - Clustering
  - Density
  - Morphology
- What are the emergent properties of security defects?
  - Building a system with secure components
- What are the reuse trade-offs?

Predictions Driving the Research Agenda



- Insiders and planted vulnerabilities control the cyber battlefield
- Predictive analysis and preemption replaces incident response as the primary security model
- Computers/Internet access replaced by numerous devices, each of which is automatically maintained
- Security shifts from a perimeter set of controls to understanding the nature of the traffic

[redacted]