Dragon Star Lecture Series (I)

Dawn Song

dawnsong@cs.berkeley.edu

Introduction

Welcome!

Staff

Instructor: Dawn Song

TA: Noah Johnson

Assistant TA: Tielei Wang (Peking Univ.)

Survey:

Name, email addr, institution, year in program, current research area (in English), general research interests (in English), suggested topics (in English), questions for instructor and TA's

Goals of Summer School

Better understanding of computer security as a field and research area

Learning state-of-the-art in selected areas

Learning how to do research

Chinese students have very good technical background & skills

Need to learn how to ask good research questions & how to innovate

Not about fish but how to fish

Schedule

Lecture Schedule: Morning

Part I: 9-9:45am, 15mins break

Part II: 10-10:45am, 15mins break

Student forum: 11-11:15am

Part III: 11:15-noon

Lunch: noon-1:30pm

Lab schedule: 1:30-5:30pm

Independent study: evening

Class Participation & Assignment

Lectures:

Interactive

Volunteer scribes (2 volunteers per segment)

Student forum:

Talk about your research and your group

3 min per individual presentation; 5 min per group presentation

Try to represent each school; weighted by FCFS

Labs: in groups

Project option

Survey option

Fri afternoon: final presentation (graded)

Tentative Syllabus (I)

Techniques and tools for vulnerability discovery and diagnosis

Taint analysis, quantitative taint analysis

Taint-based fuzzing and diagnosis

Dynamic symbolic execution

Dynamic symbolic execution for vulnerability discovery

Grammar-based symbolic execution

Loop-extended symbolic execution

Tentative Syllabus (II)

Binary analysis and its applications to computer security

Automatic patch-based exploit generation

Automatic deviation detection

Automatic vulnerability signature generation

Binary similarity analysis

Tentative Syllabus (III)

Malware analysis and defense

Botnet study

Botnet C&C reverse engineering

Automatic in-depth malware analysis

Web-based malware

Mobile malware

Tentative Syllabus (IV)

Web security

XSS vulnerability analysis and defense

JavaScript analysis

Formal foundations to web protocol security and analysis

Privacy

De-anonymization

Differential privacy

Systems implementing differential privacy and applications

Overview

What is computer security about?

Characteristics of computer security as a field of research

Adversary mindset

Highly creative, think out of the box

Highly interdisciplinary & touches every area

Overview of research areas in computer security

Outline

Vulnerability discovery

Fuzzing

Dynamic symbolic execution

Dynamic taint analysis

Taint-based fuzzing

Lab assignments

Loop-extended symbolic execution

Introduction to BitBlaze

IPhone Security Flaw

Jul 2007: “researchers at Independent Security Evaluators, said that they could take control of iPhones through a WiFi connection or by tricking users into going to a Web site that contains malicious code. The hack, the first reported, allowed them to tap the wealth of personal information the phones contain.”

Charles Miller, shown on his iPhone,

that after finding a hole in security, “you were in complete cont

iPhone attack

- **iPhone Safari downloads malicious web page**
  - **Arbitrary code is run with administrative privileges**
  - **Can read SMS log, address book, call history, other data**
  - **Can perform physical actions on the phone.**
  - **system sound and vibrate the phone for a second**
  - **could dial phone numbers, send text messages, or record**
  - **audio (as a bugging device)**
  - **Can transmit any collected data over network to attacker**

  See http://www.securityevaluators.com/iphone/

0days Are a Hacker Obsession

An 0day is a vulnerability that's not publicly known

Modern 0days often combine multiple attack vectors & vulnerabilities into one exploit

Many of these are used only once on high value targets

0day statistics

Often open for months, sometimes years

Market for 0days

Sell for $10K-100K

Tippingpoint

Eeye

Gleg.net

Dsquare

Idefense

Digital armaments

Breakingpoint

How to Find a 0day?

Step #1: obtain information

Hardware, software information

Sometimes the hardest step

- eBay to the rescue

Step #2: bug finding

Manual audit

(semi)automated techniques/tools

The iPhone Story

Step #1: WebKit opensource

svn co http://svn.webkit.org/repository/webkit/trunk WebKit

Step #2: identify potential focus points

From development site:
The JavaScriptCore Tests
"*If you are making changes to JavaScriptCore, there is an additional test suite you must rn before landing changes. This is the Mozilla JavaScript test suite*."

So we know what they use for unit testing

- Use code coverage to see which portions of code is not well tested
- Tools gcov, icov, etc., measure test coverage

Results

59.3% of 13622 lines in JavaScriptCore were covered

79.3% of main engine covered

54.7% of Perl Compatible Regular Expression (PCRE) covered

Next step: focus on PCRE

Wrote a PCRE fuzzer (20 lines of perl)

Ran it on standalone PCRE parser (pcredemo from PCRE library)

Started getting errors:
PCRE compilation failed at offset 6: internal error: code overflow

Evil regular expressions crash mobileSafari

The Art of Fuzzing

Automaticly generate test cases

Many slightly anomalous test cases are input into a target interface

Application is monitored for errors

Inputs are generally either file based (.pdf, .png, .wav, .mpg)

Or network based…

http, SNMP, SOAP

Trivial Example

Standard HTTP GET request

GET /index.html HTTP/1.1

Anomalous requests

AAAAAA...AAAA /index.html HTTP/1.1

GET ///////index.html HTTP/1.1

GET %n%n%n%n%n%n.html HTTP/1.1

GET /AAAAAAAAAAAAA.html HTTP/1.1

GET /index.html HTTTTTTTTTTTTTP/1.1

GET /index.html HTTP/1.1.1.1.1.1.1.1

Regression vs. Fuzzing

Regression: Run program on many normal inputs, look for badness.

Goal: Prevent normal users from encountering errors (e.g. assertions bad).

Fuzzing: Run program on many abnormal inputs, look for badness.

Goal: Prevent attackers from encountering exploitable errors (e.g. assertions often ok)

Approach I: Black-box Fuzz Testing

Given a program, simply feed it random inputs, see whether it crashes

Advantage: really easy

Disadvantage: inefficient

Input often requires structures, random inputs are likely to be malformed

Inputs that would trigger a crash is a very small fraction, probability of getting lucky may be very low

Enhancement I: Mutation-Based Fuzzing

Take a well-formed input, randomly perturb (flipping bit, etc.)

Little or no knowledge of the structure of the inputs is assumed

Anomalies are added to existing valid inputs

Anomalies may be completely random or follow some heuristics (e.g. remove NUL, shift character forward)

Examples:

E.g., ZZUF, very successful at finding bugs in many real-world programs, http://sam.zoy.org/zzuf/

Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.

Example: fuzzing a pdf viewer

Google for .pdf (about 1 billion results)

Crawl pages to build a corpus

Use fuzzing tool (or script to)

1. Grab a file
2. Mutate that file
3. Feed it to the program
4. Record if it crashed (and input that crashed it)

Mutation-based Fuzzing In Short

Strengths

Super easy to setup and automate

Little to no protocol knowledge required

Weaknesses

Limited by initial corpus

May fail for protocols with checksums, those which depend on challenge response, etc.

Enhancement II: Generation-Based Fuzzing

Test cases are generated from some description of the format: RFC, documentation, etc.

Using specified protocols/file format info

E.g., SPIKE by Immunity
http://www.immunitysec.com/resources-freesoftwa

Anomalies are added to each possible spot in the inputs

Knowledge of protocol should give better results than random fuzzing

Example: Protocol Description

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Generation-Based Fuzzing In Short

Strengths

completeness

Can deal with complex dependencies e.g. checksums

Weaknesses

Have to have spec of protocol

- Often can find good tools for existing protocols e.g. http, SNMP

Writing generator can be labor intensive for complex protocols

The spec is not the code

Fuzzing Tools

Hackers' job made easy

Input generation

Input injection

Bug detection

Workflow automation

Input Generation

Existing generational fuzzers for common protocols (ftp, http, SNMP, etc.)

Mu-4000, Codenomicon, PROTOS, FTPFuzz

Fuzzing Frameworks: You provide a spec, they provide a fuzz set

SPIKE, Peach, Sulley

Dumb Fuzzing automated: you provide the files or packet traces, they provide the fuzz sets

Filep, Taof, GPF, ProxyFuzz, PeachShark

Many special purpose fuzzers already exist as well

ActiveX (AxMan), regular expressions, etc.

Input Injection

Simplest

Run program on fuzzed file

Replay fuzzed packet trace

Modify existing program/client

Invoke fuzzer at appropriate point

Use fuzzing framework

e.g. Peach automates generating COM interface fuzzers

Bug Detection

See if program crashed

Type of crash can tell a lot (SEGV vs. assert fail)

Run program under dynamic memory error detector (valgrind/purify)

Catch more bugs, but more expensive per run.

See if program locks up

Roll your own checker e.g. valgrind skins

Workflow Automation

Sulley, Peach, Mu-4000 all provide tools to aid setup, running, recording, etc.

Virtual machines can help create reproducable workload

How Much Fuzz Is Enough?

Mutation based fuzzers may generate an infinite number of test cases... When has the fuzzer run long enough?

Generation based fuzzers may generate a finite number of test cases. What happens when they're all run and no bugs are found?

Example: PDF

I have a PDF file with 248,000 bytes

There is one byte that, if changed to particular values, causes a crash

This byte is 94% of the way through the file

Any single random mutation to the file has a probability of .00000392 of finding the crash

On average, need 127,512 test cases to find it

At 2 seconds a test case, thats just under 3 days...

It could take a week or more...

Code Coverage

Some of the answers to these questions lie in *code coverage*

Code coverage is a metric which can be used to determine how much code has been executed.

Data can be obtained using a variety of profiling tools. e.g. `gcov`

Types of Code Coverage

Line/block coverage

Measures how many lines of source code have been executed.

Branch coverage

Measures how many branches in code have been taken (conditional jmps)

Path coverage

Measures how many paths have been taken

Example

```
if( a > 2 )
a = 2;
if( b > 2 )
b = 2;
```

Requires

1 test case for line coverage

2 test cases for branch coverage

4 test cases for path coverage

- i.e. `(a,b) = {(0,0), (3,0), (0,3), (3,3)}`

Code Coverage

Benefits:

How good is this initial file?

Am I getting stuck somewhere?

```
if(packet[0x10] < 7) { //hot path
} else { //cold path

}
```

How good is fuzzer X vs. fuzzer Y

Am I getting benefits from running a different fuzzer?

Problems:

Code can be covered without revealing bugs

```
mySafeCpy(char *dst, char*
src){
        if(dst && src)
            strcpy(dst, src);
    }
```

Fuzzing Rules of Thumb

Protocol specific knowledge very helpful

Generational tends to beat random, better spec's make better fuzzers

More fuzzers is better

Each implementation will vary, different fuzzers find different bugs

The longer you run, the more bugs you may find

Best results come from guiding the process

Notice where your getting stuck, use profiling!

Code coverage can be very useful for guiding the process

Can we do better?

Approach II: Constraint-based Automatic Test Case Generation

Look inside the box

Use the code itself to guide the fuzzing

Assert security/safety properties

Explore different program execution paths to check for security properties

Challenge:

1. For a given path, need to check whether an input can trigger the bug, i.e., violate security property
2. Find inputs that will go down different program execution paths

Running Example

```
f(unsigned int len){
  unsigned int s;
  char *buf;
    if len % 2==0;
    then s = len;
    else s = len + 2;
  buf = malloc(s);
  read(fd, buf, len);
   …
}
```

Where's the bug?

What's the security/safety property?

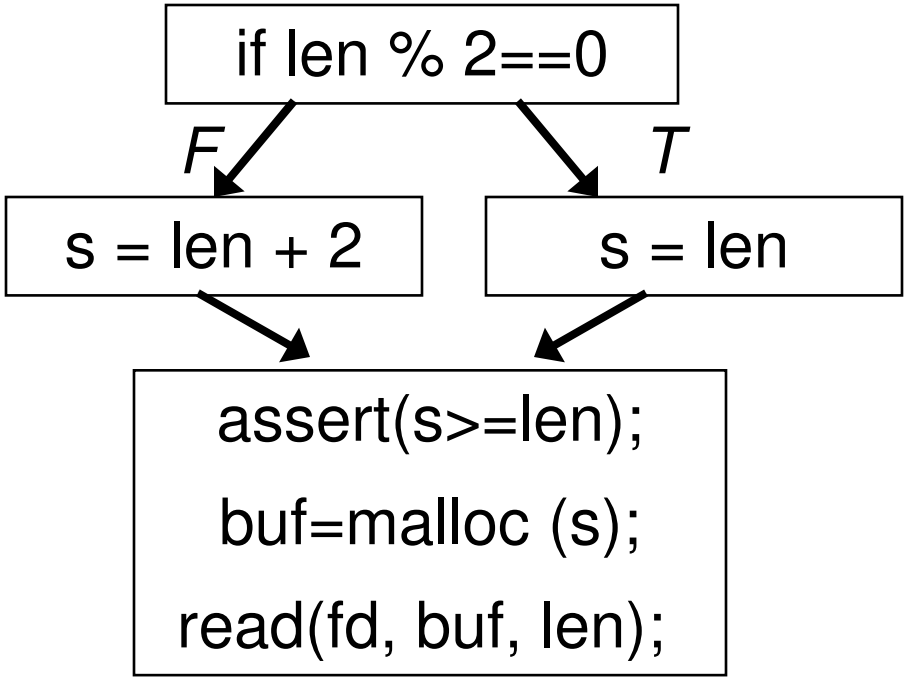
s>=len

What inputs will cause violation of the security property?

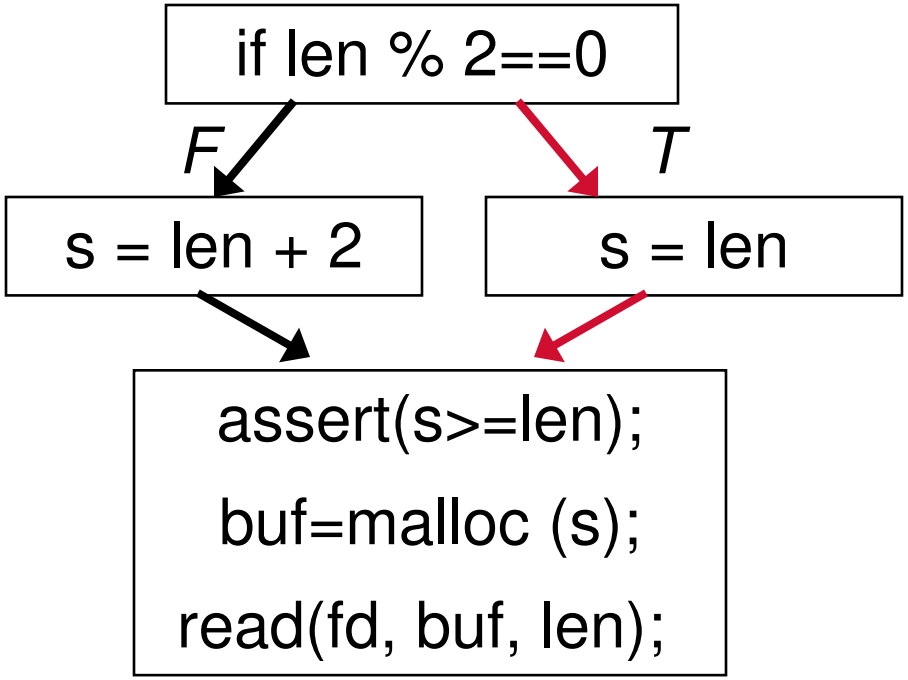
len = 232 - 1

How likely will random testing find the bug?

Running Example



Symbolic Execution

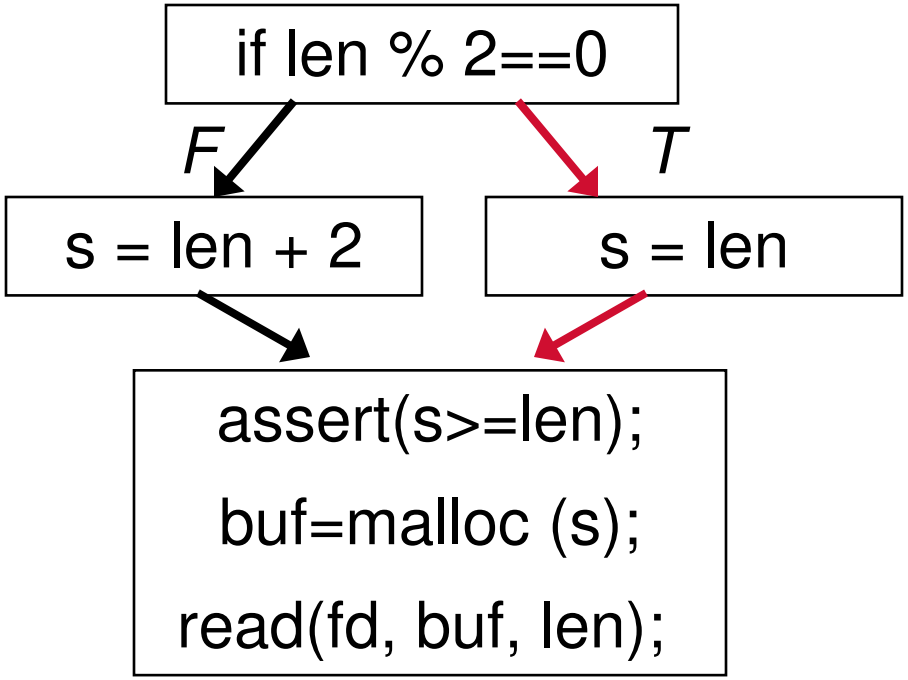


Test input len=6

No assertion failure

What about all inputs that takes the same path as len=6?

Symbolic Execution



What about all inputs that takes the same path as len=6?

Represent len as symbolic variable

Symbolic Execution

Represnet inputs as symbolic variables

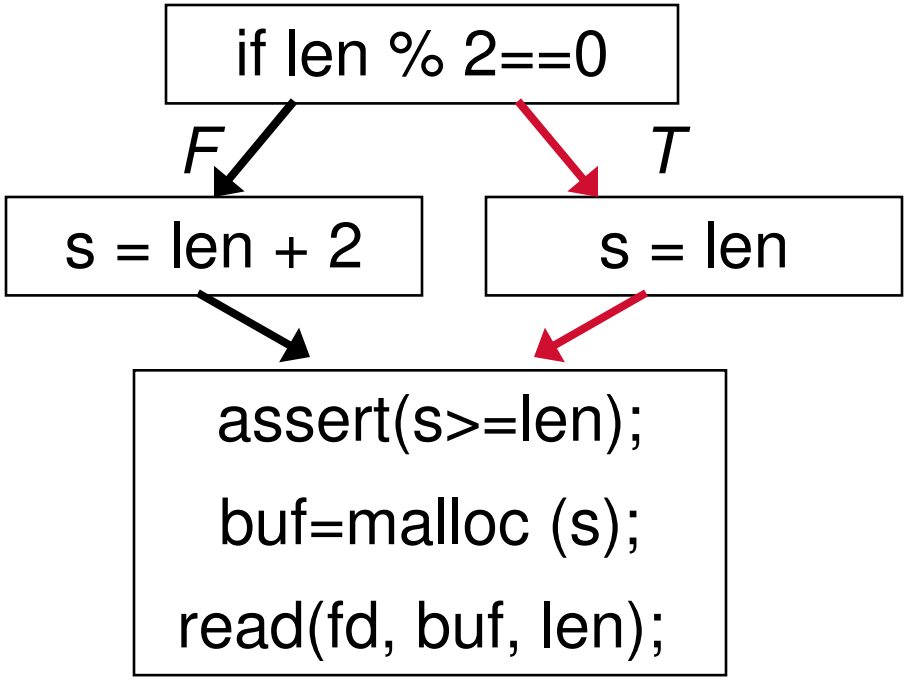
Perform each operation on symbolic variables symbolically

x = y + 5;

Registers and memory values dependent on inputs become symoblic expressions

Certain conditions for conditional jump become symbolic expressions as well

Symbolic Execution



len % 2 = 0 (path constraint)

s = len

s<len?

What about all inputs that takes the same path as len=6?

Represent len as symbolic variable

Using a Solver

Is there a value for len s.t.
len % 2 = 0 ^ s = len ^ s < len?

Give the symbolic formula to a solver

In this case, the solver returns No

The formula is not satisfiable

What does this mean?

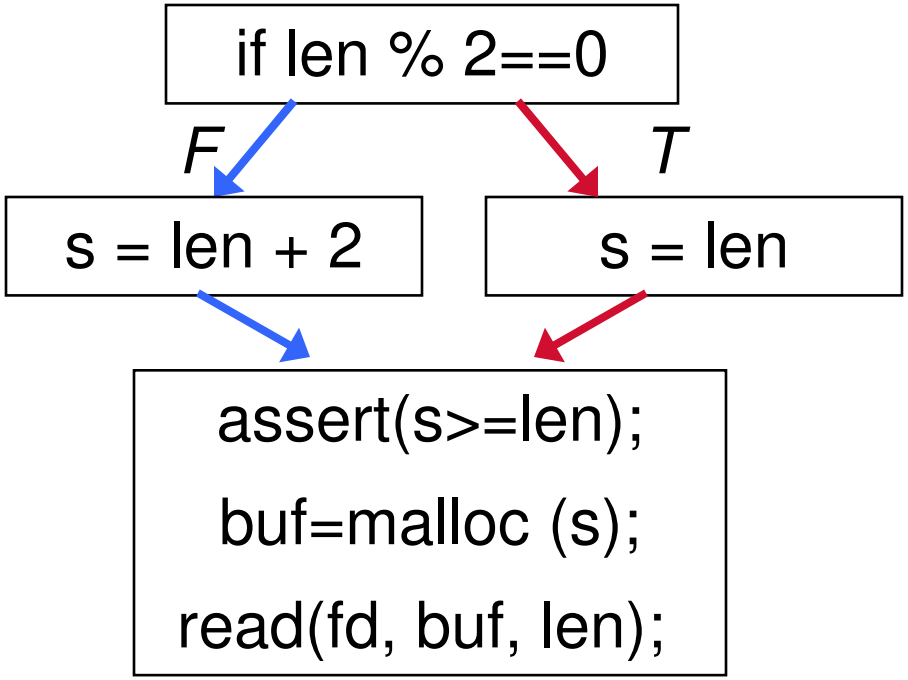
For any len that follows the same path as len = 6,
the execution will be safe

Symbolic execution can check many inputs at the same time for the same path

What to do next?

Try to explore different path

How to Explore Different Paths?



Previous path constraint: len % 2 = 0

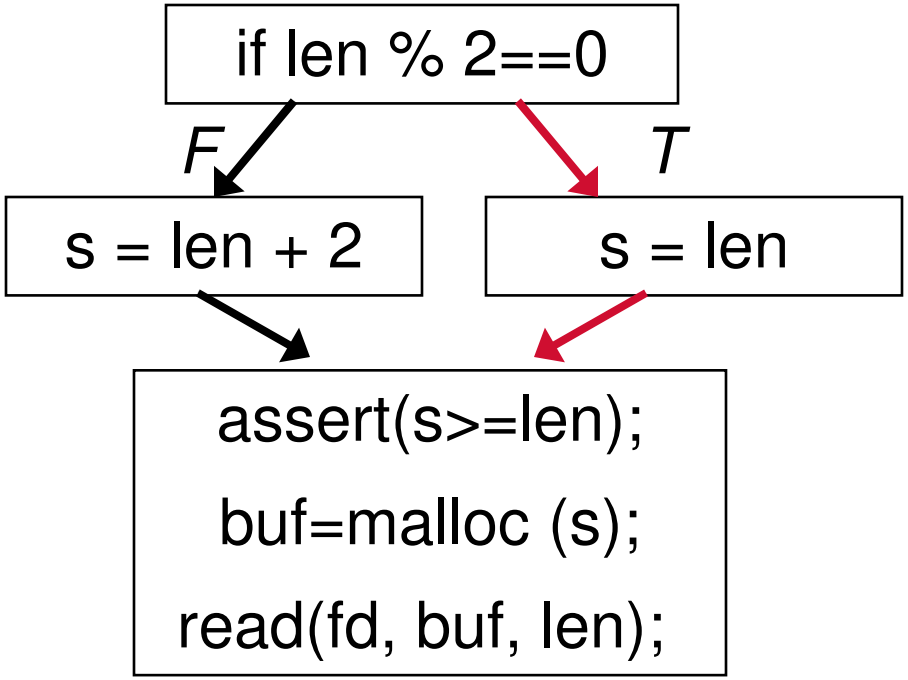
Flip the branch to go down a different path:

len % 2 != 0

Using a solver for the formula

A satisfying assignment is a new input to go down the path

Checking Assertion in the Other Path



len % 2 != 0 (path constraint)

s = len + 2

s<len?

- **Is there a value for len s.t. len % 2 != 0 ^ s = len+2 ^ s < len?**
- **Give the symbolic formula to a solver**
  - **Solver returns satisfying assignment: len = 232 -1**
  - **Found the bug!**

Summary: Symbolic Execution for Bug Finding

Symbolicly execute a path

Create the formula representing:
path constraint ^ assertion failure

Give the solver the formula

- If returns a satisfying assignment, a bug found

Reverse condition for a branch to go down a different path

Give the solver the new path constraint

If returns a satisfying assignment

- The path is feasible
- Found a new input going down a different path

Pioneer work

EXE, DART, CUTE

Towards Next Generation of BitBlaze

Dawn Song

Computer Science Dept.

UC Berkeley

Viruses

Worms

Botnets

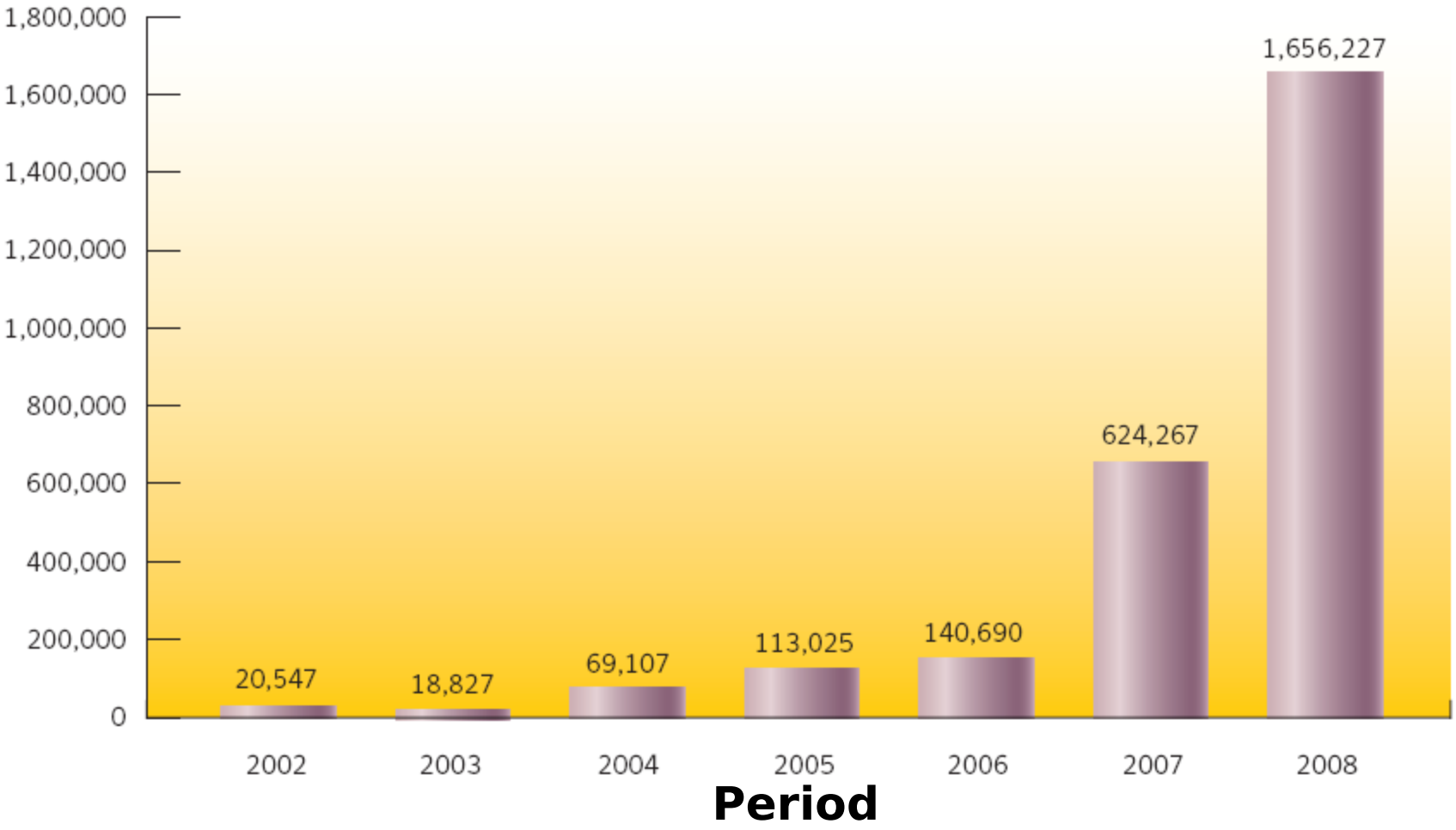
Trojan Horses

Rootkits

Spyware

Malicious Code: Critical Threat

Growth of New Malicious Code Threats



(source: Symantec)

Viruses

Worms

Botnets

Trojan Horses

Rootkits

Spyware

Malicious Code: Critical Threat

Defense is Challenging

Software inevitably has bugs/security vulnerabilities

Intrinsic complexity

Time-to-market pressure

Legacy code

Long time to produce/deploy patches

Attackers have real financial incentives to exploit them

Thriving underground market

Large scale zombie platform for malicious activities

Attacks increase in sophistication

We need more effective techniques and tools for defense

Previous approaches largely symptom & heuristics based

The BitBlaze Approach & Research Foci

v Semantics based, focus on root cause:
Automatically extracting security-related properties from binary code for effective vulnerability detection & defense

1. Build a unified binary analysis platform for security

Identify & cater common needs of different security applications

Leverage recent advances in program analysis, formal methods, binary instrumentation/analysis techniques for new capabilities

2. Solve real-world security problems via binary analysis

- Extracting security related models for vulnerability detection
- Generating vulnerability signatures to filter out exploits
- Dissecting malware for forensics & offense: e.g., botnet infiltration
- More than a dozen security applications & publications

ze: Computer Security via Program Binary Ana

- Unified platform to accurately analyze security properties of binaries
  - Security evaluation & audit of third-party code
  - Defense against morphing threats
  - Faster & deeper analysis of malware

Detecting Vulnerabilities

Generating Filters

Dissecting Malware

BitBlaze Binary Analysis Infrastructure

BitBlaze Binary Analysis Infrastructure: Challenges

Important to handle binary-only setting

COTS & malicious code scenarios

Binary is truthful

Complexity

IA-32 manuals for x86 instruction set weights over 11 pounds

Lack higher-level semantics

Even disassembling is non-trivial

Require whole-system view

Operations within kernel and interactions btw processes

Malicious code may obfuscate

Code packing

Code encryption

Code obfuscation & dynamically generated code

BitBlaze Binary Analysis Infrastructure: Design Rationale

Accuracy

Enable precise analysis, formally modeling instruction semantics

Extensibility

Develop core utilities to support different architecture and applications

Fusion of static & dynamic analysis

Static analysis

- Pros: more complete results
- Cons: pointer aliasing, indirect jumps, code obfuscation, kernel & floating point instructions difficult to model

Dynamic analysis

- Pros: easier
- Cons: limited coverage

Solution: combining both

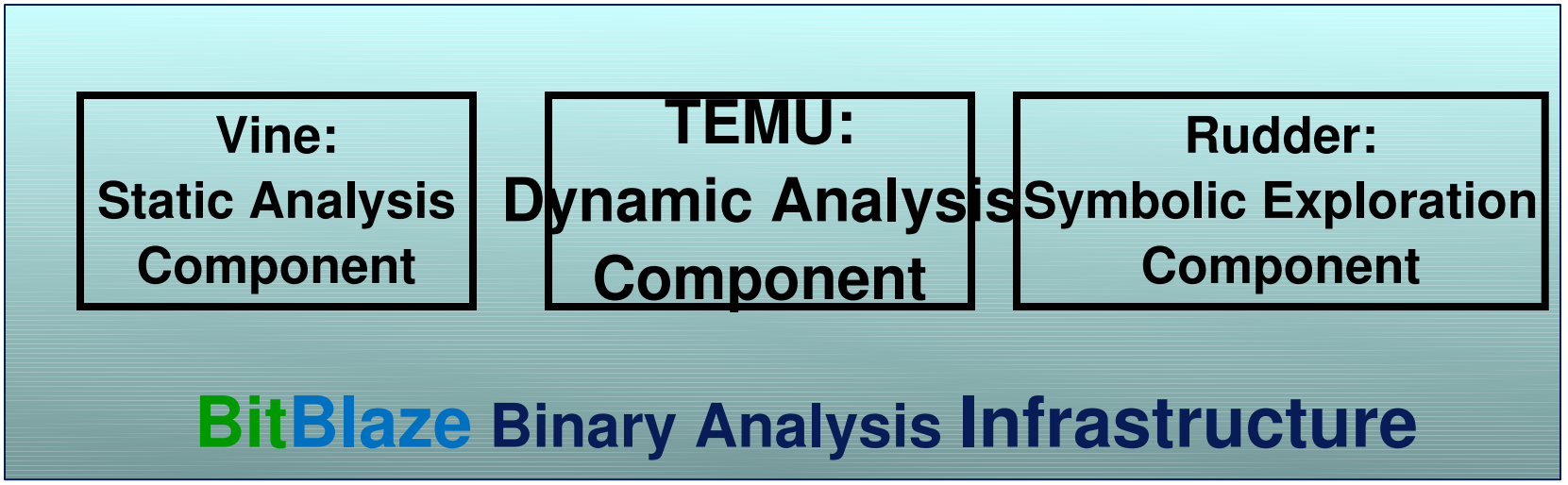
BitBlaze Binary Analysis Infrastructure: Architecture

The first infrastructure:

Novel fusion of static, dynamic, formal analysis methods

Whole system analysis (including OS kernel)

Analyzing packed/encrypted/obfuscated code



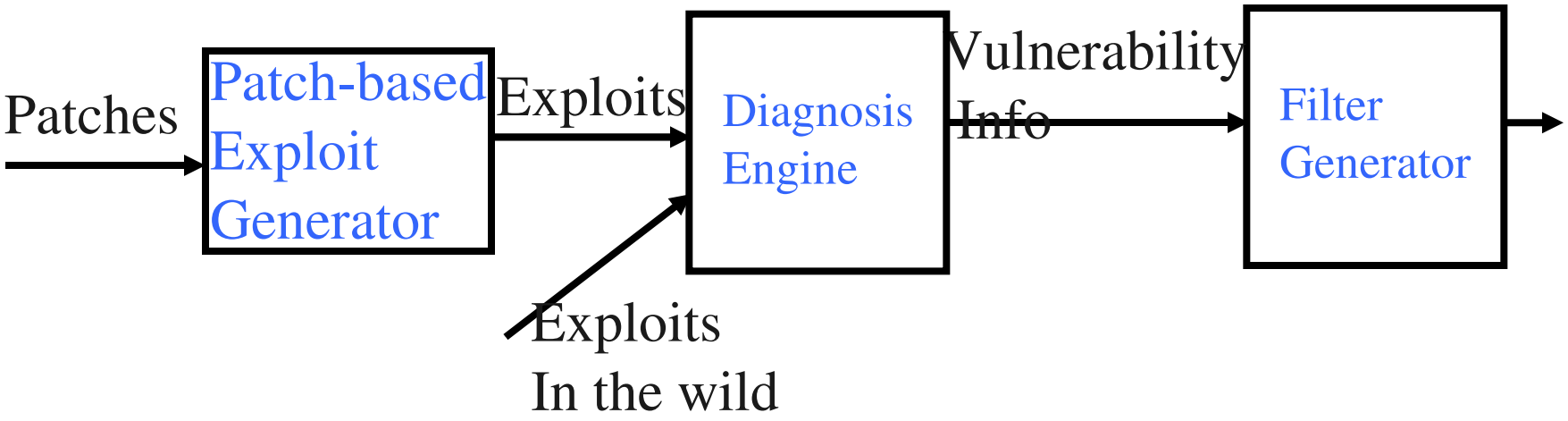
BitBlaze in Action: Addressing Security Problems

Effective new approaches for diverse security problems

Over dozen projects

Over 12 publications in security conferences

Exploit generation, diagnosis, defense



In-depth malware analysis

Others: reverse engineering, deviation detection, etc..

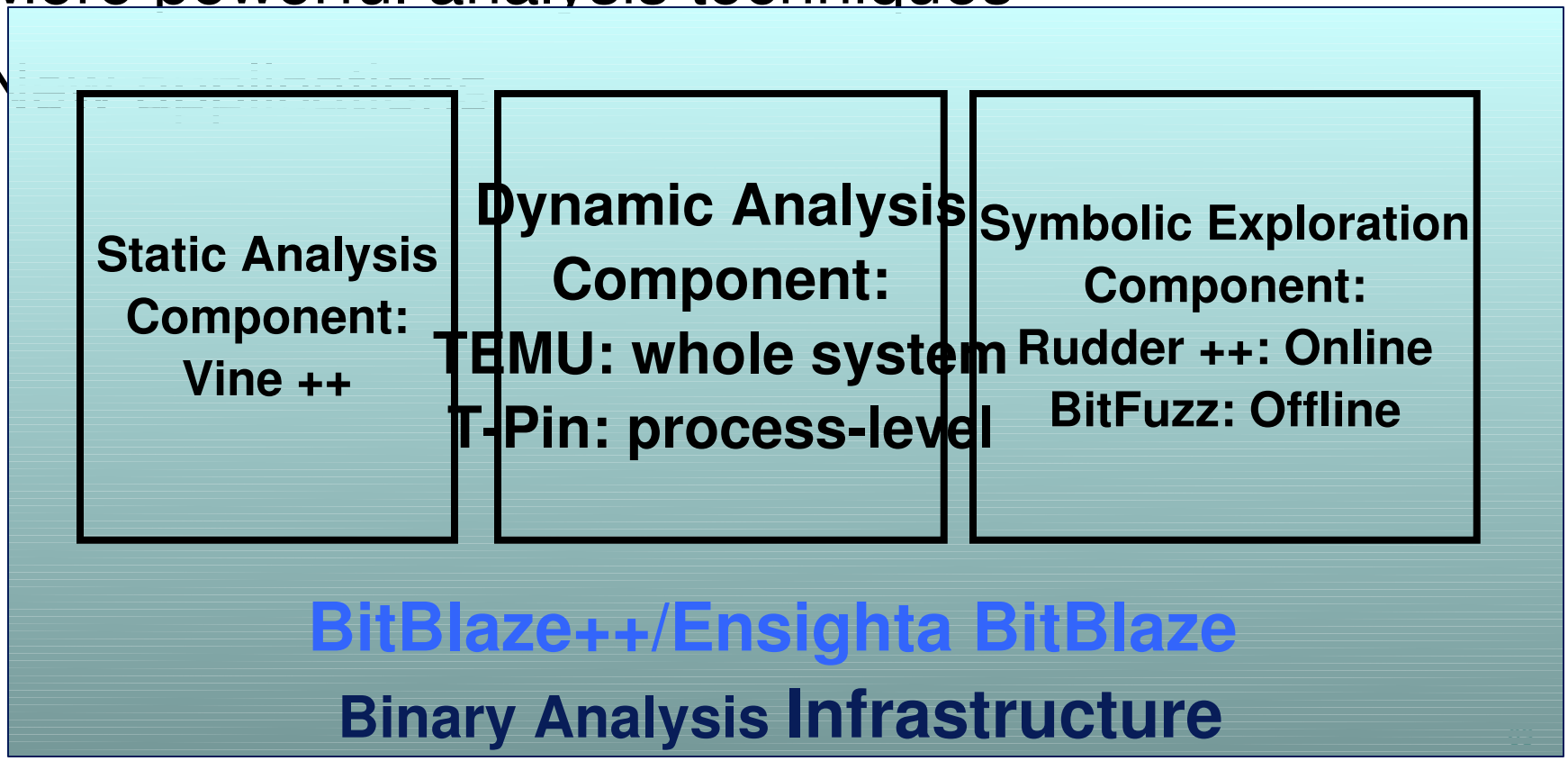
Towards Next Generation of BitBlaze (I)

BitBlaze++/Ensighta BitBlaze

Better scalability

More powerful analysis techniques

N



Towards Next Generation of BitBlaze (II)

Symbolic reasoning is key enabler to many applications in BitBlaze

Vulnerability discovery and diagnosis

Vulnerability filter generation

In-depth malware analysis

Limitations of previous dynamic symbolic execution

Difficult to handle loops

Difficult to handle complex encoding functions

Difficult to inputs with complex grammar

Need to start from beginning of program, difficult to reach deep

More powerful analysis techniques for symbolic reasoning

Loop-extended symbolic execution

Decomposition-&-re-stitching symbolic execution

Grammar-based symbolic exploration

On-the-spot symbolic execution

Outline

Motivation

Summary of BitBlaze

Overview of BitBlaze++/Ensighta BitBlaze

New symbolic exploration techniques

Loop extended symbolic execution

Decomposition-&-re-stitching symbolic execution

Grammar-based symbolic exploration

On-the-spot symbolic execution

Other new capabilities

Binary code reuse

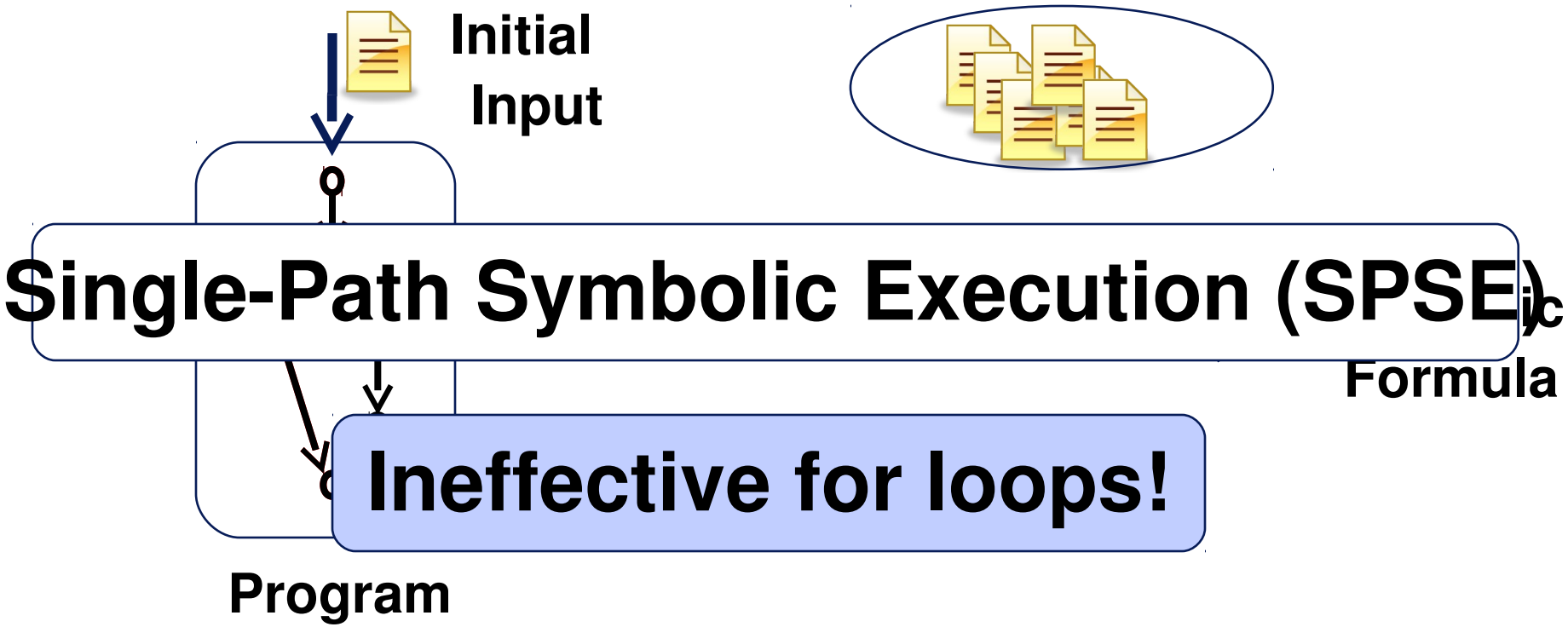
Automatic protocol reverse engineering

Conclusion

Dynamic Symbolic Execution

- **Combines concrete execution with symbolic execution**
- **Automatically explore program execution space**
- **Has important applications**
  - **Program Testing and Analysis**
  - **Automatic test case generation**
  - **Given an initial test case, find a variant that executes a different path**
  - **Computer Security**
  - **Vulnerability Discovery & Exploit Generation**
  - **Given an initial benign test case, find a variant that triggers a bug**
  - **Vulnerability Diagnosis & Signature Generation**
  - **Given an initial exploit for a vulnerability, find a set of conditions necessary to trigger it**

Limitations of Previous Approach

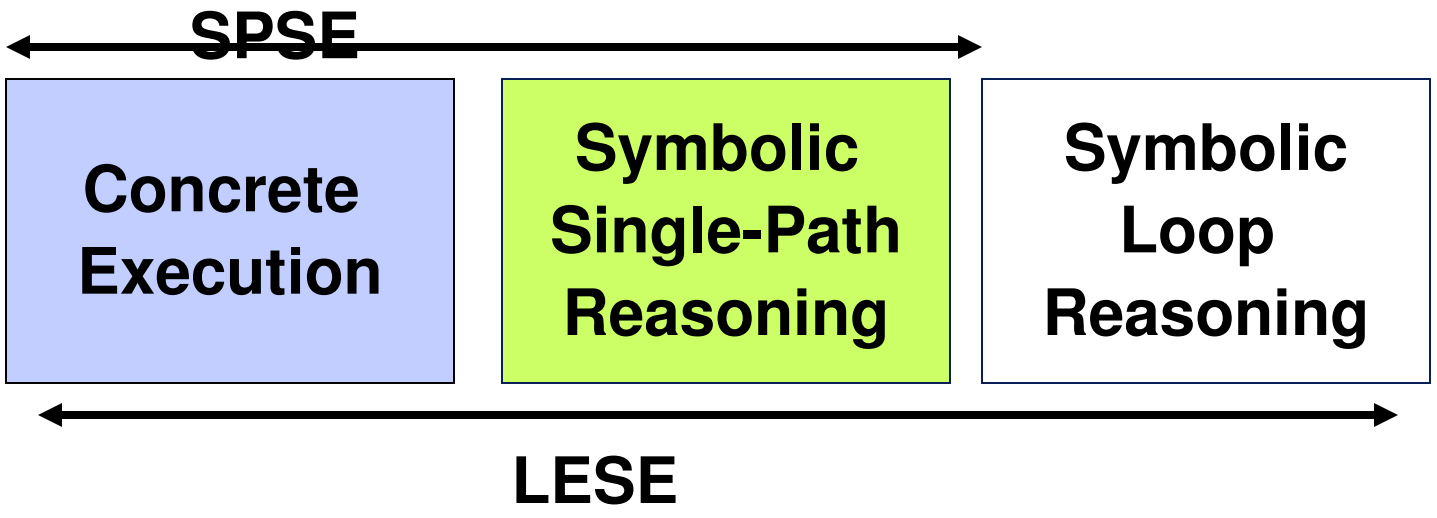


Concrete Execution

Symbolic Execution

Contributions of Our Work

- **Loop-Extended Symbolic Execution (LESE)**
  - **Generalizes symbolic reasoning to loops**



- **Applicable directly to binaries**
- **Demonstrate its effectiveness in an important security application**
  - **Buffer overflow diagnosis & discovery**
  - **Show scalability for practical real-world examples**

Motivation: A HTTP Server Example


```
void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++)    {
        URL [i] = input [p++];
                }
}
```

Calculating length

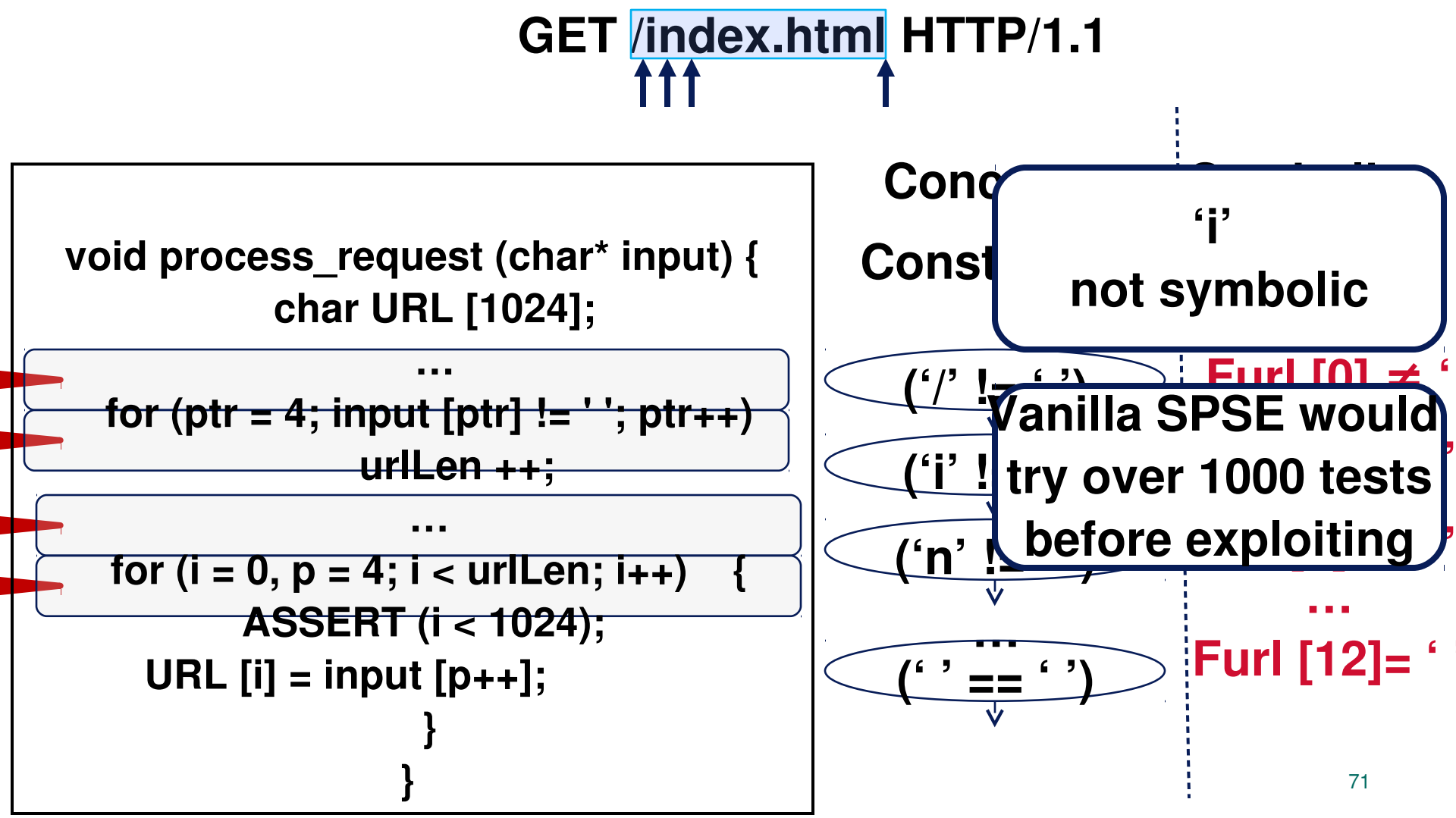
Copying URL to buffer

Motivation: A HTTP Server Example

- **Goal: Check if the buffer can be overflowed**

```
void process_request (char* input) {
        char URL [1024];
        …
  for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
        …
  for (i = 0, p = 4; i < urlLen; i++)    {
        ASSERT (i < 1024);
    URL [i] = input [p++];
        }
  }
```

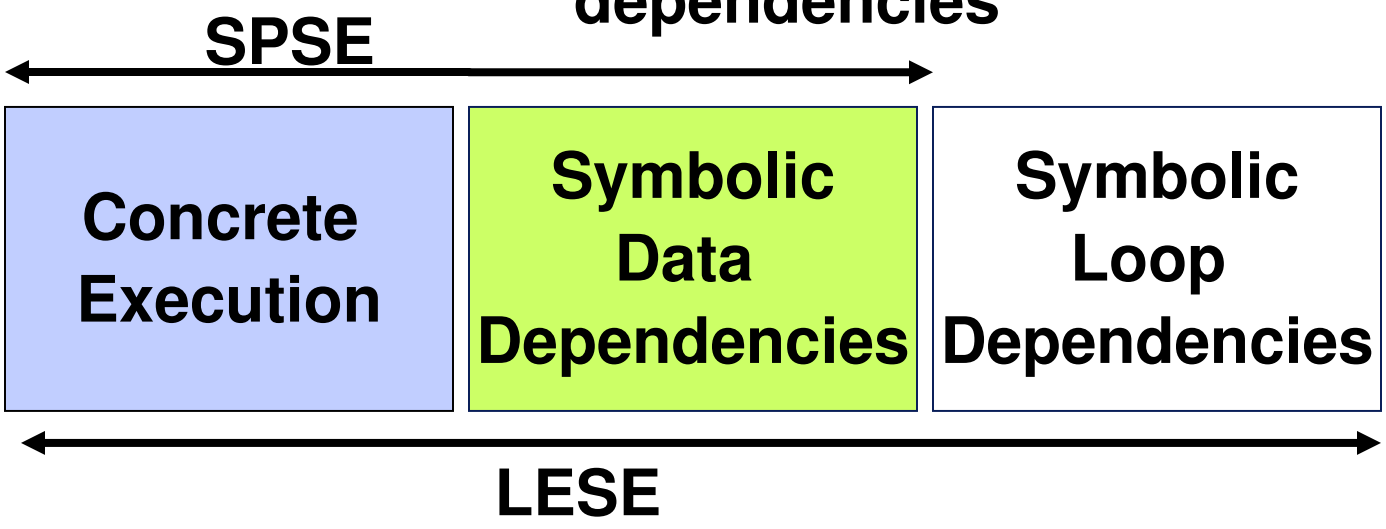
Motivation: A HTTP Server Example



Intuition

- **LESE: Finds an exploit for the example in 1 step**
  - **Key Point: Summarize loop effects**

- **Intuition: Why was 'i' not symbolic?**
  - **SPSE only tracks *data dependencies***
  - **'i' was loop dependent**
- **Model loop-dependencies in addition to data dependencies**



Our Approach

Introduce a symbolic “trip count” for each loop

Symbolic variable representing the number of times a loop executes

LESE has 2 steps

STEP 1: Derive relationship between program variables and trip counts

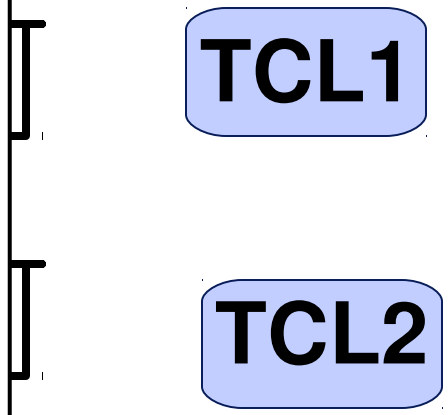
- Linear Relationships

STEP 2: Relate trip counts to inputs

Introducing Symbolic Trip Counts

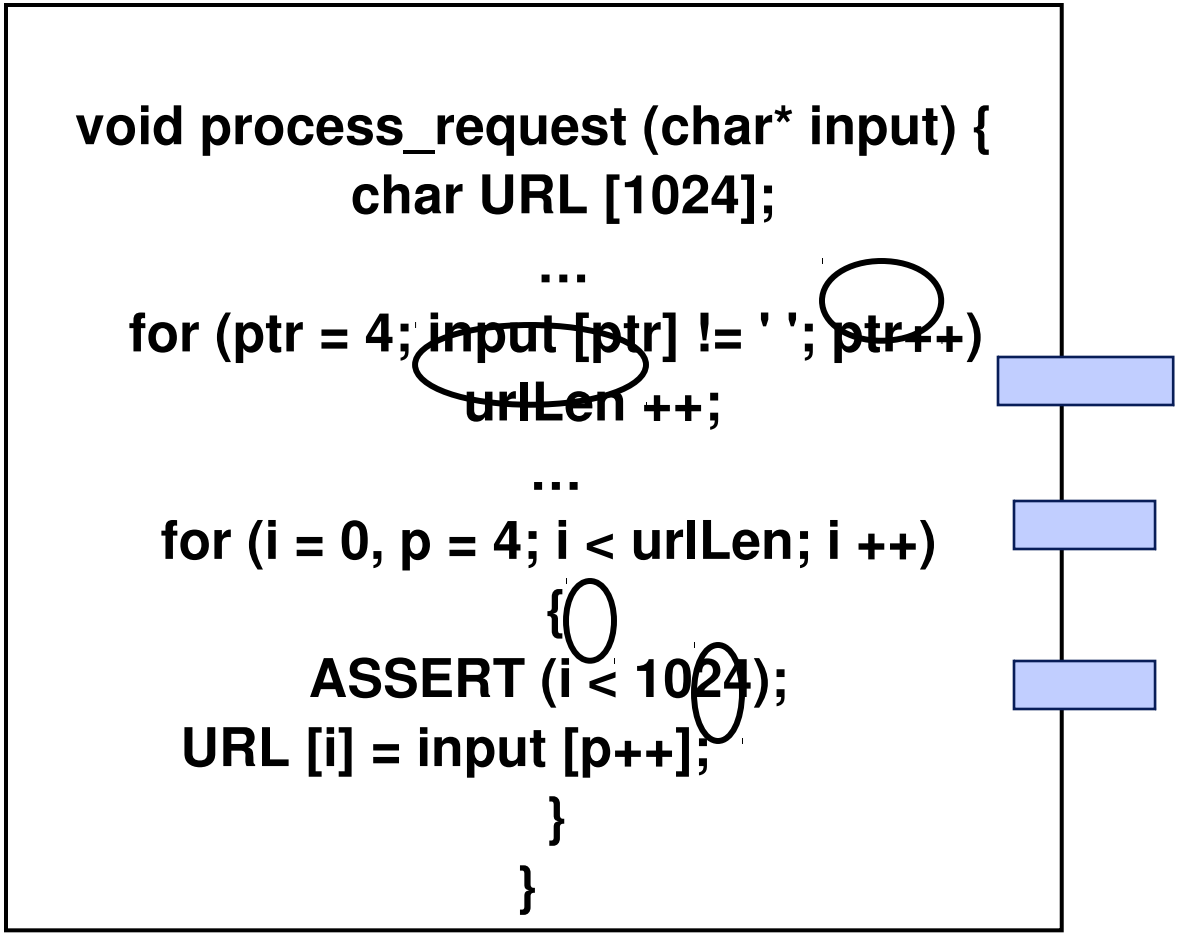
Introduces symbolic loop *trip counts*

```
void process_request (char* input) {
    char URL [1024];
    ...
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    ...
    for (i = 0, p = 4; i < urlLen; i++)    {
        ASSERT (i < 1024);
        URL [i] = input [p++];
    }
}
```



Step 1: Relating program variables to TCs

Links trip counts to program variables



Symbolic Constraints

ptr = 4 + TCL1
urlLen = 0 + TCL1

(i < urlLen)

i = -1 + TCL2
p = 4 + TCL2

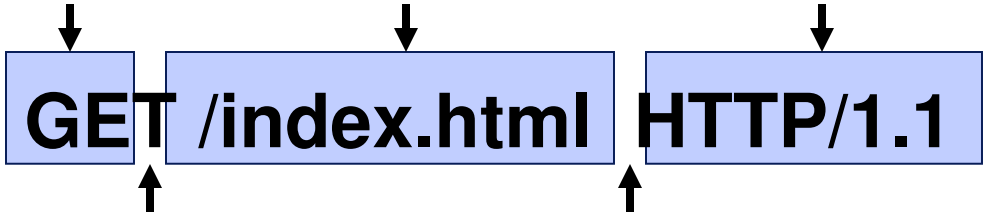
Step 2: Relating Trip Counts to Input

Inputs

Initial Concrete Test Case

A Grammar

- Fields
- Delimiters



Implicitly models symbolic attributes for fields

Lengths of fields

Counts of repeated elements

Available from off-the-shelf tools

Network application grammars in Wireshark, GAPA

Media file formats in Hachoir, GAPA

Can even be automatically inferred [CCS07,S&P09]

Step 2: Link trip counts to input

Link trip counts to the input grammar

```
void process_request (char* input) {
    char URL [1024];
    …
  for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    …
  for (i = 0, p = 4; i < urlLen; i++)    {
        ASSERT (i < 1024);
    URL [i] = input [p++];
                }
            }
```

Symbolic Constraints

(Furi [0] ≠ ‘ ’) && (Furi [1] ≠ ‘ ’)
&&
… (Furi [12] == ‘ ’)

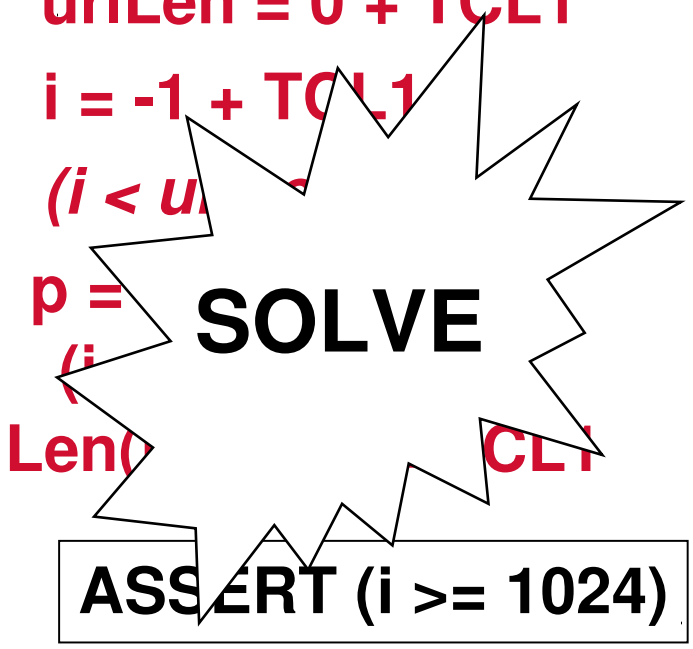
Len(FURL) == TCL1

Solve using a decision procedure

Link trip counts to the input grammar

```
void process_request (char* input) {
    char URL [1024];
    …
    for (ptr = 4; input [ptr] != ' '; ptr++)
        urlLen ++;
    …
    for (i = 0, p = 4; i < urlLen; i++)    {
        ASSERT (i < 1024);
        URL [i] = input [p++];
    }
}
```

Symbolic Constraints



Solution: HTTP Server Example

Solve constraints

```
void process_request (char* input) {
    char URL [1024];
    …
  for (ptr = 4; input [ptr] != ' '; ptr++)
      urlLen ++;
    …
  for (i = 0, p = 4; i < urlLen; i++)    {
      ASSERT (i < 1024);
    URL [i] = input [p++];
        }
    }
```

Exploit Condition

Len(FURL) > 1024

GET aaa..

(1025 times)…

Challenges

Problems:

Identifying loop dependencies on binaries

- Syntactic induction variable analysis insufficient

Capturing the inter-dependence between two loops

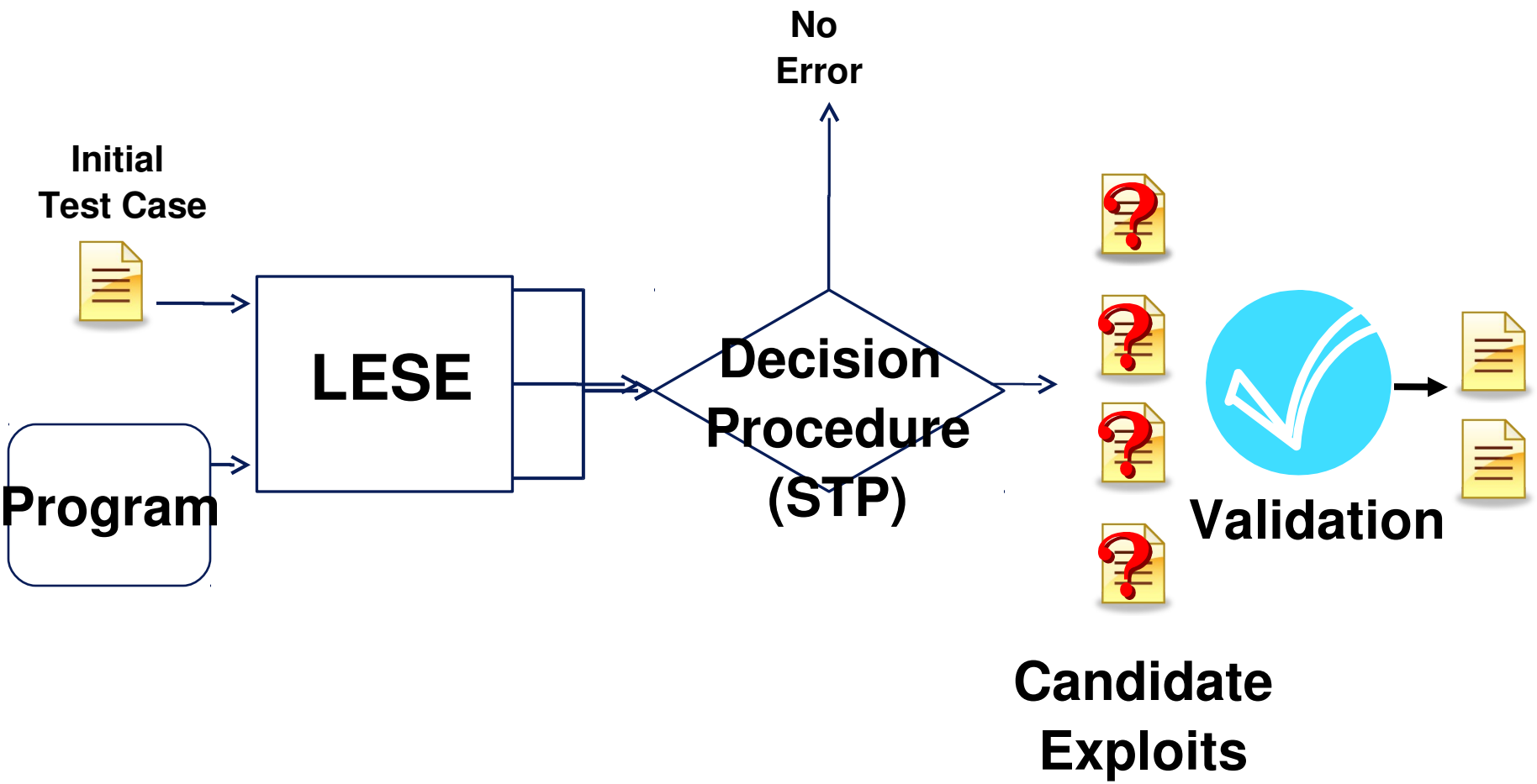
- An induction variable of may influence trip counts of subsequent loops

Our Solution

Dynamic abstract interpretation of x86 machine code

Reason about inter-dependence

Experimental Setup



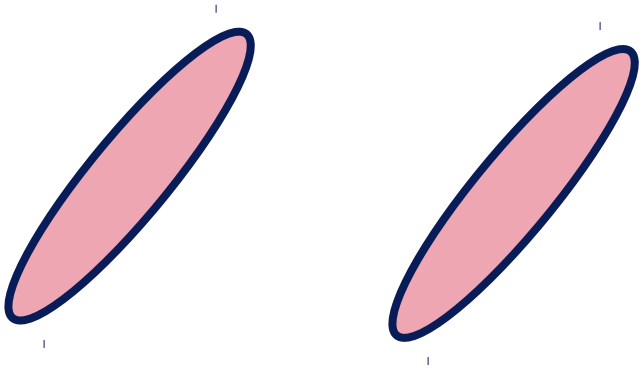
Results (I): Vulnerability Discovery

On 14 benchmark applications (MIT Lincoln Labs)

Created from historic buffer overflows (BIND, sendmail, wuftp)

Found 1 or more vulnerabilities in each benchmark

1 new exploit location in sendmail 7 benchmark



Results (II): Real-world Vulnerabilities

Diagnosis and Discovery 3 Real-world Case Studies

SQL Server Resolution [Slammer Worm 2003]

GDI Windows Library [MS07-046]

Gaztek HTTP web Server

Diagnosis Results

Results precise and field level

Dis

car

1 n

Program	Buffer size (bytes)	Condition for overflow
GHttpd (1)	220	`URI.len > 172`
GHttpd (2)	208	`URI.len > 133`
SQL Server	128	`DBName.len > 64`
GDI	4096	`(2·INP[19:18])»2 < 0`

Results (III): Loop statistics

Identifies new symbolic conditions

Loop Conditions

LESE Summary

LESE is a generalization of SPSE

Captures effect of program inputs on loops

Summarizes the effect of loops on program variables

Works for real-world Windows and Linux binaries

Key enabler for several applications

Buffer overflow discovery and diagnosis

- Capable of finding new bugs
- Does not require manual function summaries