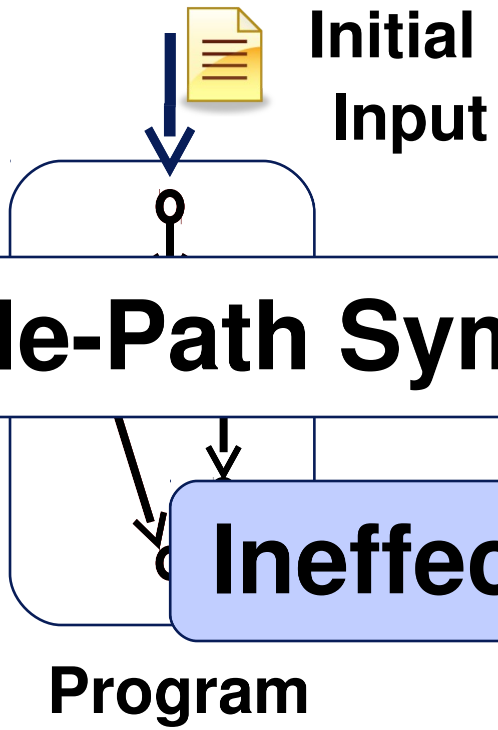
Dynamic Symbolic Execution

- **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



Single-Path Symbolic Execution (SPSE)

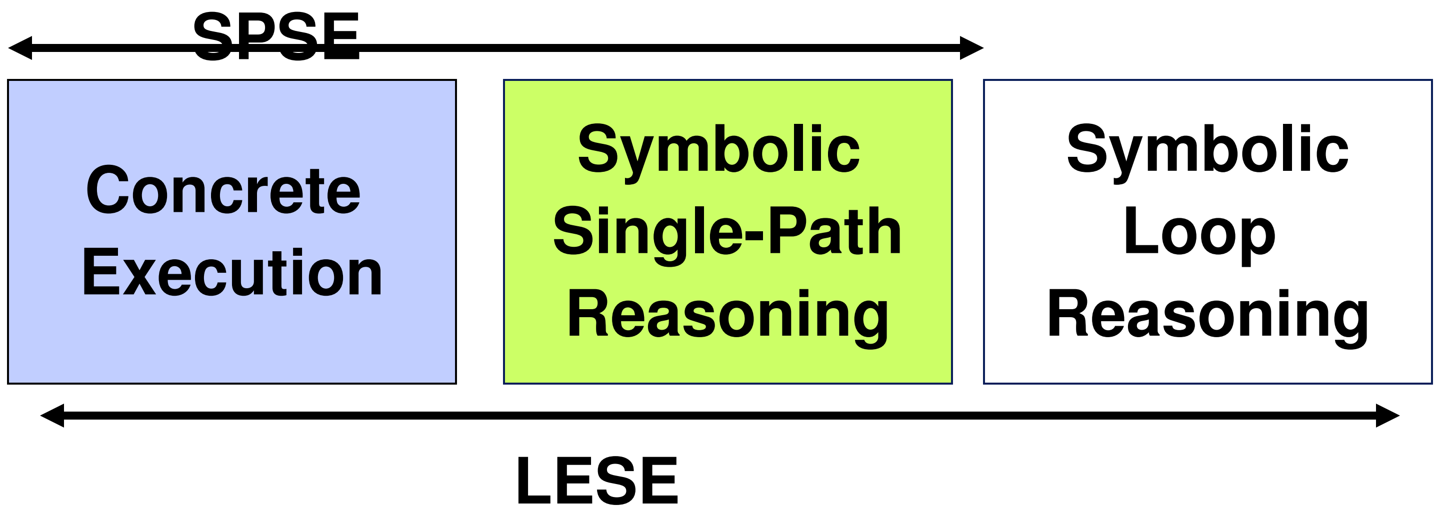
Ineffective for loops!

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

GET /index.html HTTP/1.1

CMD — **URL** — **VERSION**

```
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

GET /index.html HTTP/1.1

```
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++];
    }
}
```

Conc

Const

‘i’
not symbolic

(‘/’ != ‘ ’)

(‘i’ !

(‘n’ !

...

(‘ ’ == ‘ ’)

Vanilla SPSE would try over 1000 tests before exploiting

Furl [0] ≠ ‘

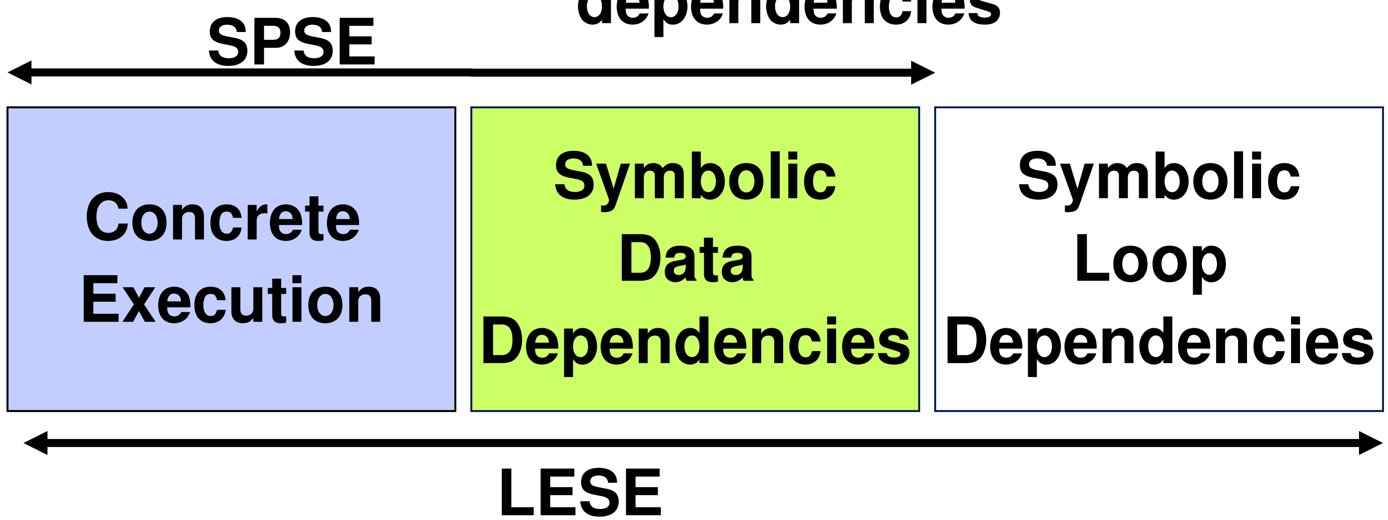
...

Furl [12]= ‘

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++];
    }
}
```

TCL1

TCL2

Step 1: Relating program variables to TCs

Links trip counts to program variables

```
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

ptr = 4 + TCL1
urlLen = 0 + TCL1

(i < urlLen)

i = -1 + TCL2
p = 4 + TCL2

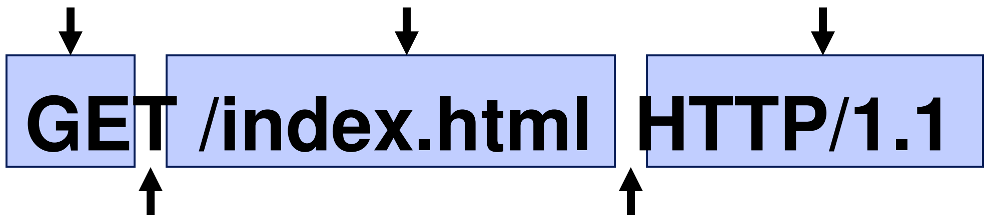
Step 2: Relating Trip Counts to Input

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] == ‘ ’)

G

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

ptr = 4 + TCL1

urlLen = 0 + TCL1

i = -1 + TCL1

(i < u

p =

(i

Len(CL1

SOLVE

ASSERT (i >= 1024)

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

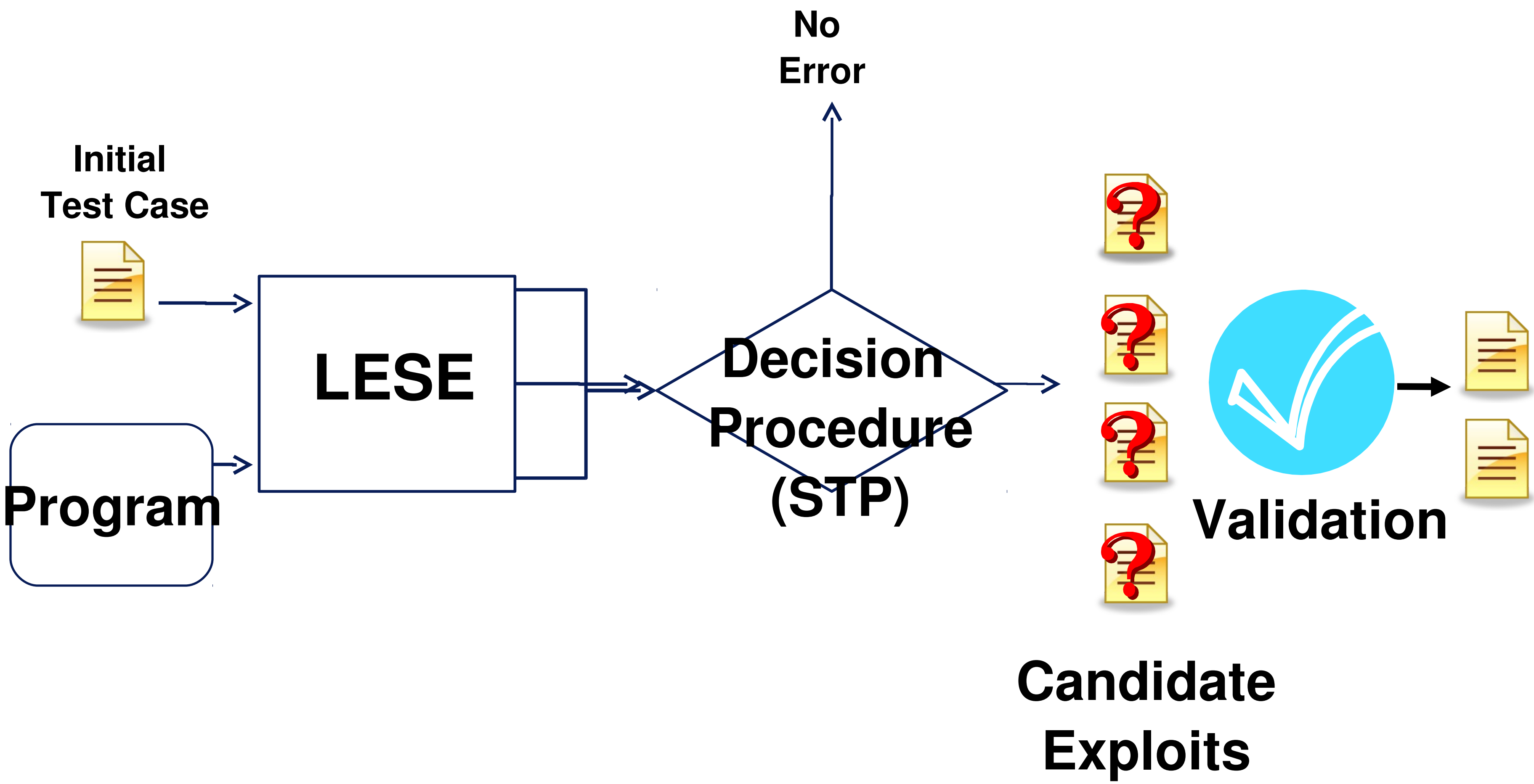
- An induction variable of may influence trip counts of subsequent loops

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

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

Problem

Dynamic symbolic execution important for bug finding

But, fails on programs that use encoding functions

Decryption, decompression, checksum, hash

Encoding functions introduce complex constraints

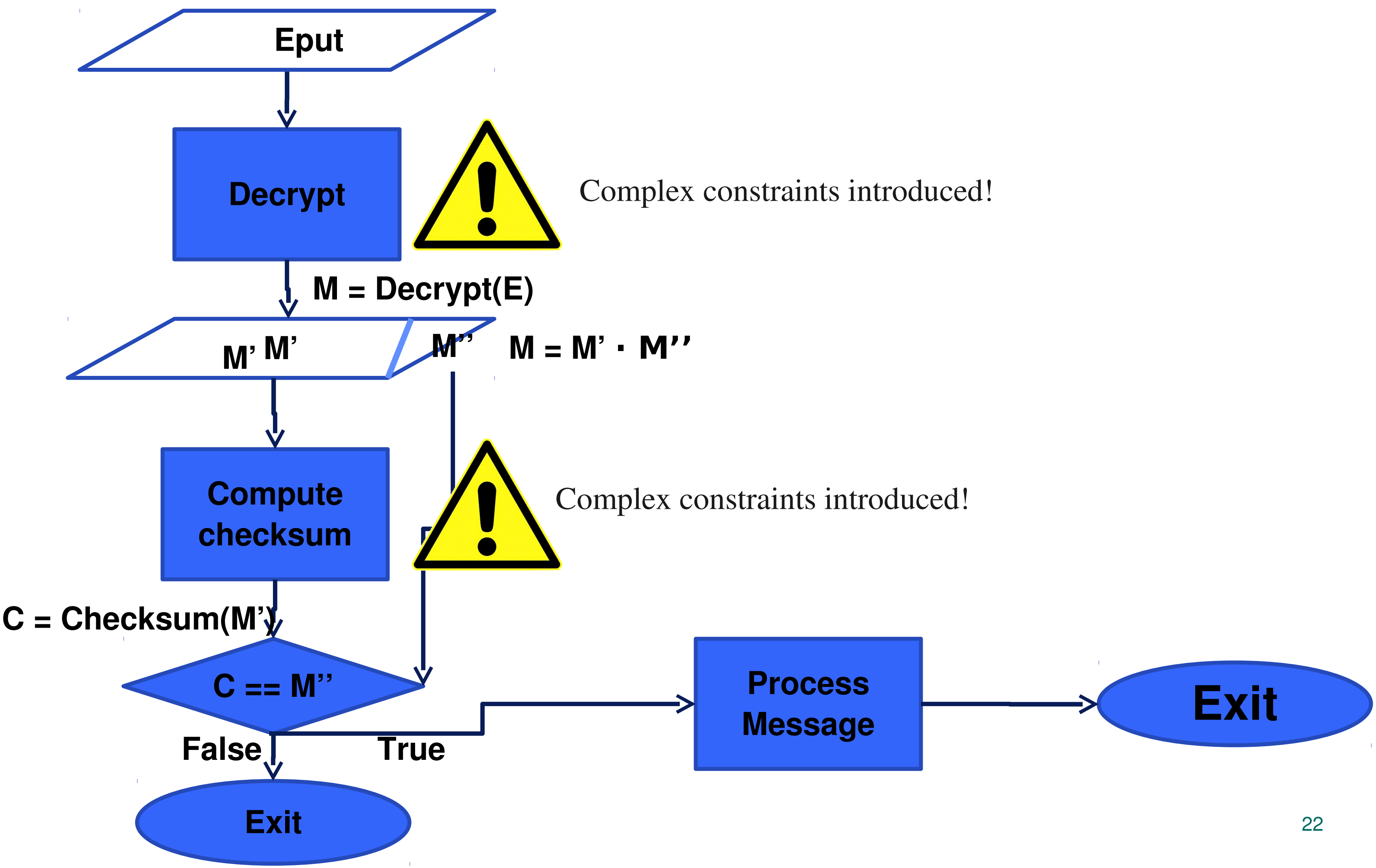
Solver faces constraints designed to be complex

e.g., cryptographic hash: SHA1, MD5

Similar problems for other bug finding techniques

Taint-based fuzzing, Grammar-aware fuzzing…

Program



Eput
Decrypt
Complex constraints introduced!
M = Decrypt(E)
M’ M’
M’’
M = M’ · M’’
Compute checksum
Complex constraints introduced!
C = Checksum(M’)
C == M’’
False
True
Process Message
Exit
Exit

Decomposition + Re-Stitching

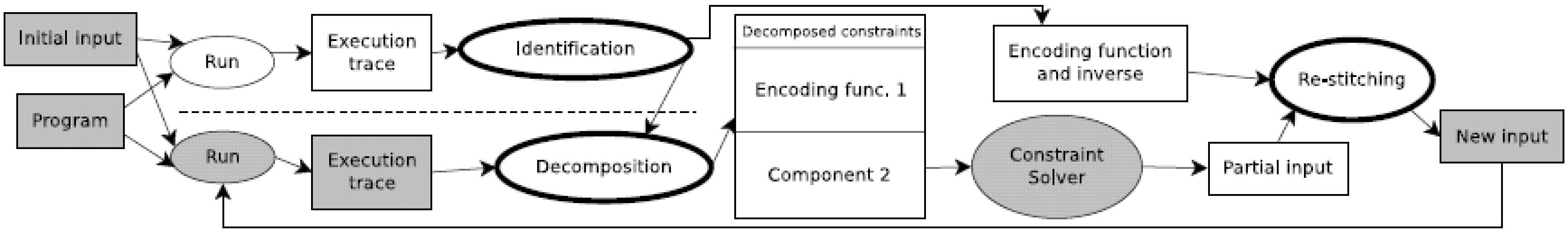
Compositional approach

Break execution into phases: encoding(s) + rest

Two types of decomposition

1. Serial (e.g., decryption)
   2. Surjective transformation (input not used afterwards)
   3. Create new symbols on output of encoding function
4. Side-condition (e.g., checksum)
   5. Can be satisfied by changing another part of the input
   6. Remove symbols from output of encoding function

Approach



- **Exploration is an iterative process**
- **Three stages:**
  1. **Identify encoding functions (done once)**
  2. **Output identification**
  3. **Includes inverse functions (e.g., encryption)**
  4. **Decompose path predicate (in each iteration)**
  5. **Re-stitch to create a new input**

Application

Finding bugs in malware

Potential applications

Cleaning hosts

Malware genealogy

Cyberwarfare

Many ethical, legal issues need to be addressed

We show that the technical issues can be addressed

We wish to start a discussion on the use of these bugs

Results: Stitched vs. Vanilla

Compare Stitched vs. Vanilla explorations

Run both on same malware for 10 hours and find bugs

Name	Vulnerability Type	Encoding function	Search Time (Stitched)	Search Time (Vanilla)
Zbot	Null dereference	checksum	17.8 sec	>600 min
Zbot	Infinite loop	checksum	129.2 sec	>600 min
MegaD	Process	decryptio	8.5 sec	>600 min

Results: Bug reproducibility

Each malware family comprises many binaries over time

Packing, functionality changes …

Bugs have been present in malware families for long time

Name	Number of Binaries	Bug reproducibility	Newest	Oldest
MegaD	4	~2 years	Feb. 24, 2010	Feb. 22, 2008
Gheg	5	~9.5 months	Nov. 28,	Feb. 6,

Towards Next Generation of BitBlaze

Dawn Song

Computer Science Dept.

UC Berkeley

Viruses

Worms

Botnets

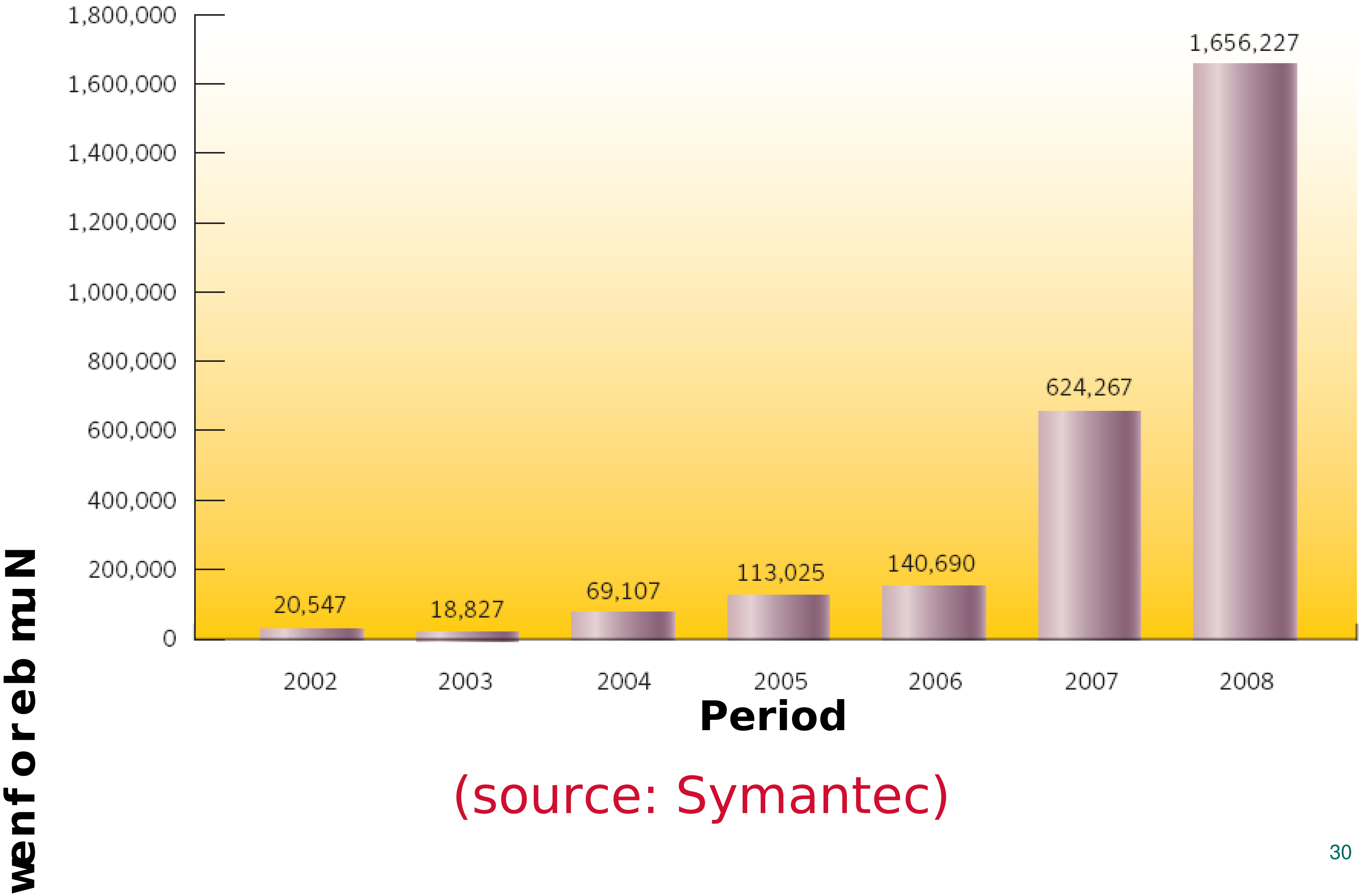
Trojan Horses

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

- 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

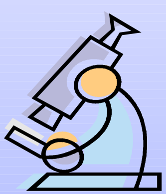
2. Solve real-world security problems via binary analysis

   - 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

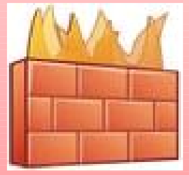
ze: Computer Security via Program Binary Ana

- 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

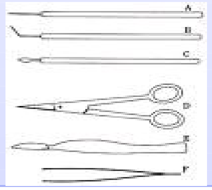
Detecting Vulnerabilities



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

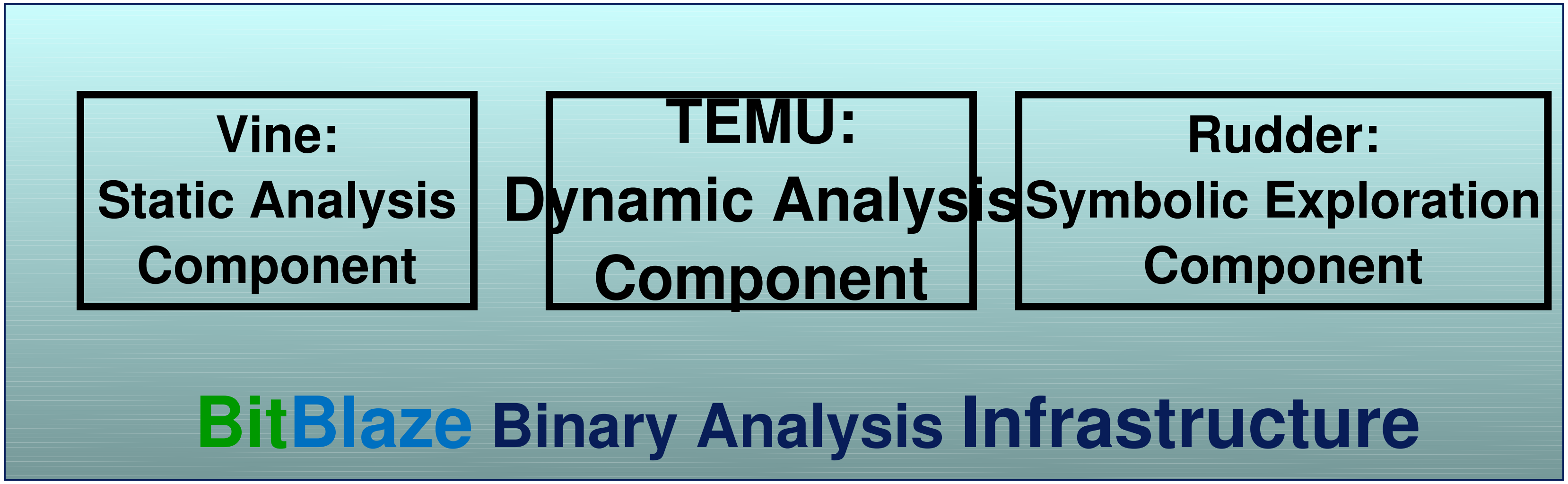
BitBlaze Binary Analysis Infrastructure: Architecture

The first infrastructure:

Novel fusion of static, dynamic, formal analysis methods

Whole system analysis (including OS kernel)

Analyzing packed/encrypted/obfuscated code



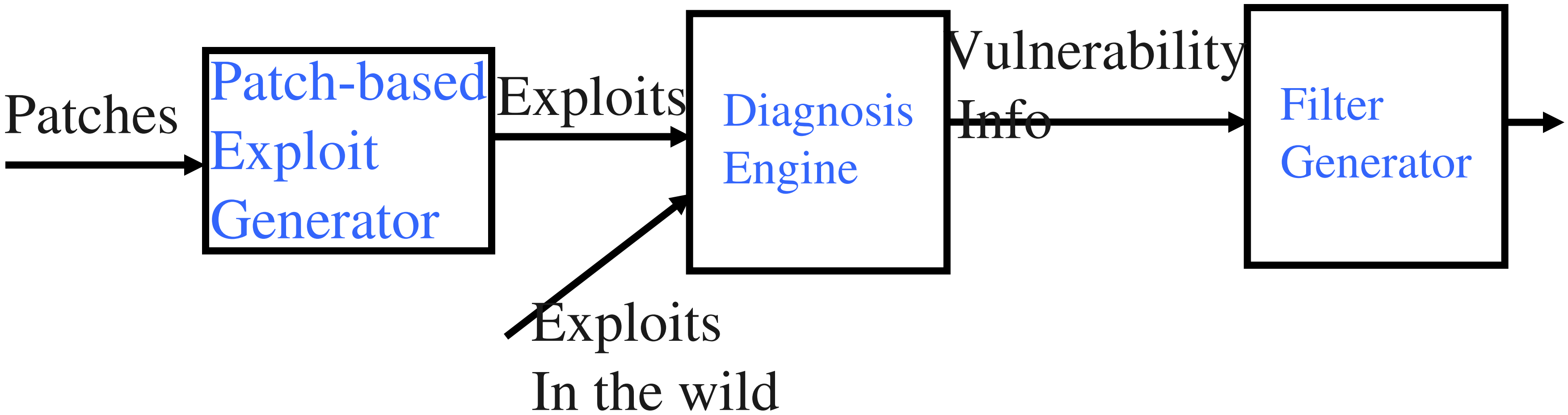
BitBlaze in Action: Addressing Security Problems

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..

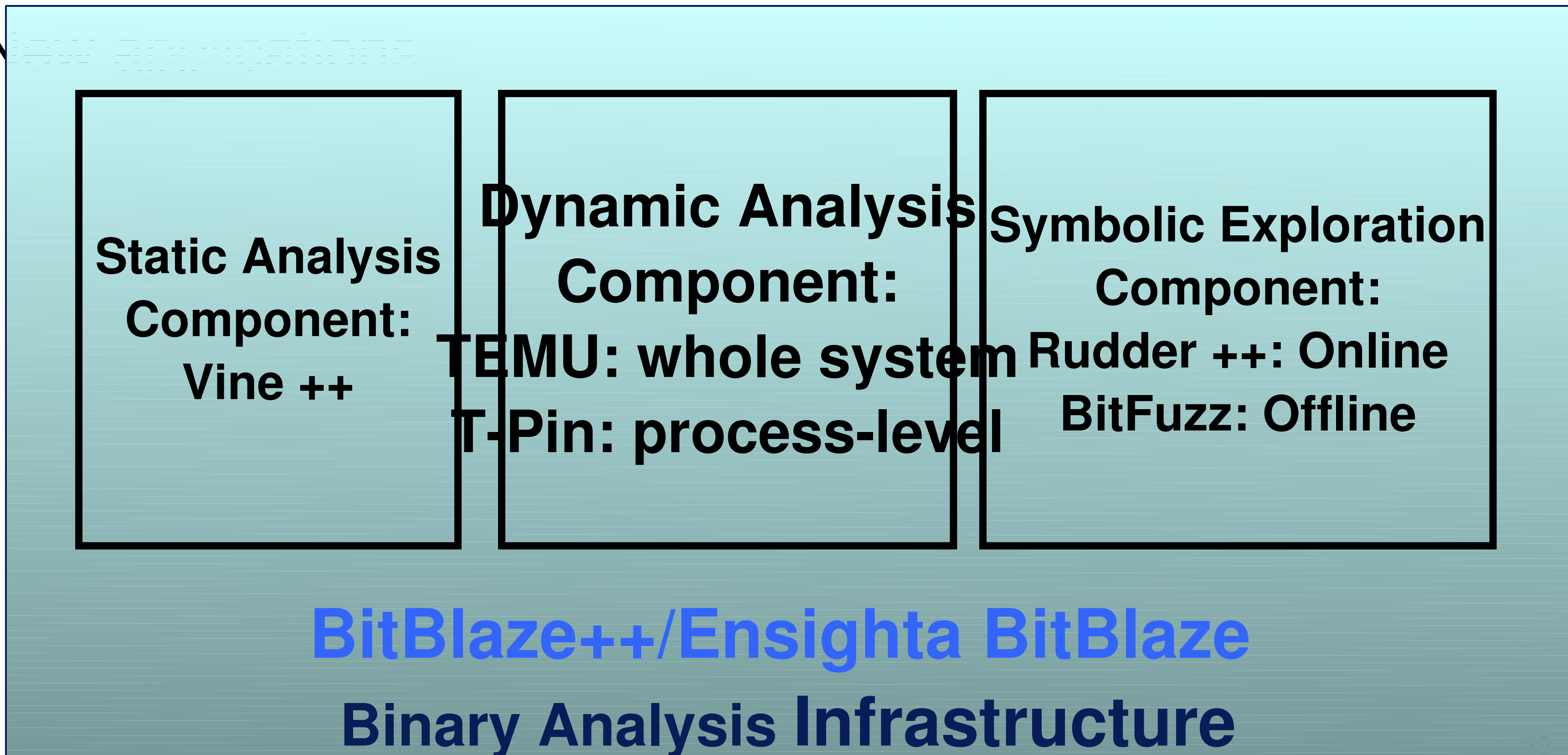
Towards Next Generation of BitBlaze (I)

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



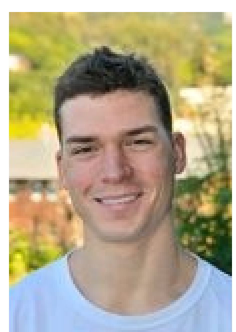


bitblaze.cs.berkeley.edu

webblaze.cs.berkeley.edu

dawnsong@cs.berkeley.edu





Logistics

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

Forming groups:

2-3 people per group

Lab:

Project option

- Proposal due tomorrow night
- 2-page report

Survey option

- Proposal/topic due tonight
- 5-page report

Student Forum

Abstract submission: title, name, institution, abstract