Dynamic Analysis and Debugging of Binary Code for Security Applications

Lixin (Nathan) Li
Senior Research Lead
Cyber Innovation Unit
Battelle Memorial Institute, USA

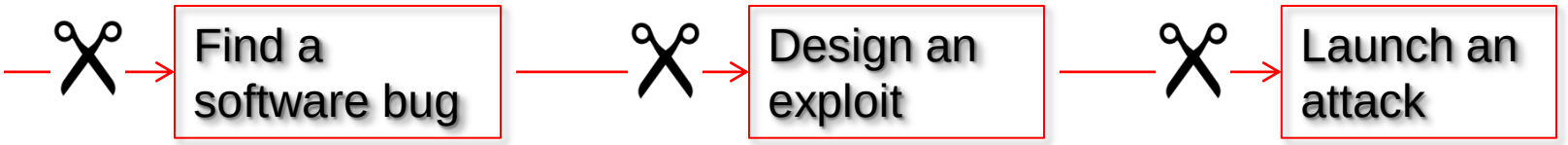
Chao Wang
Assistant Professor
Dept. Electrical & Computer Engr.
Virginia Tech, USA

Battelle
The Business of Innovation

VirginiaTech
Invent the Future

Why bother?

The vast majority of security vulnerabilities in today's cyber systems originate from improper software implementations.



Program analysis and verification techniques are critical during this process

Why dynamic analysis?

Static program analysis is a great technique.

Analyzing the (C/C++) source code is also important.

However, in security applications, it is important to perform dynamic analysis of the binary code.

Why interactive analysis?

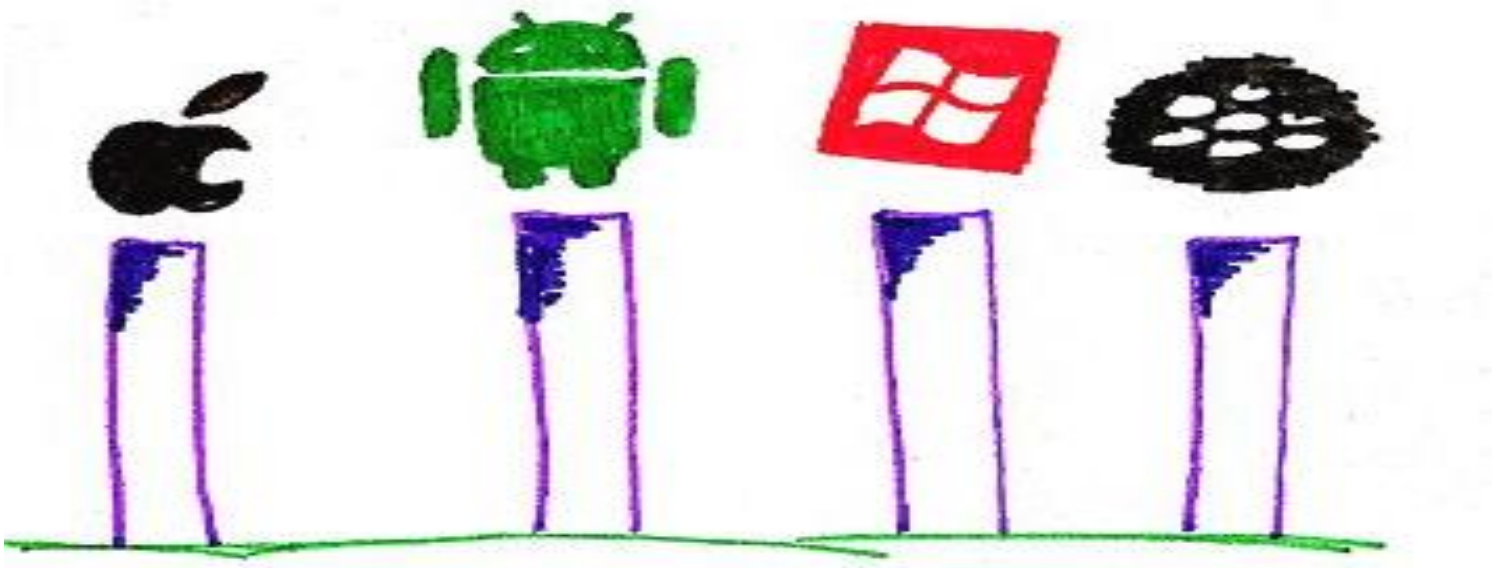
Security applications often require "human in the loop" analysis – they are very difficult to be fully automated.

However, tools support is severely lacking.

We want to build new software tools that can be used to speed up the interactive analysis.

Why cross-platform analysis?

- **Platform-dependent** Execution Trace Generation
  - Intermediate Representation(**IR**) Translation
- **Platform-independent** Analysis
  - Reusing the algorithm and implementation

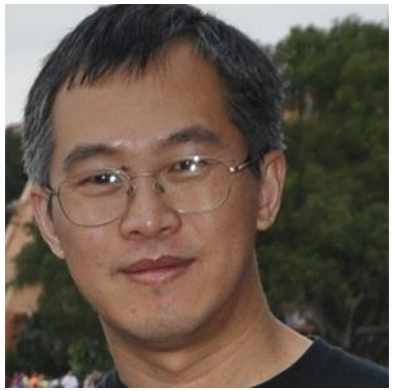


Security Problem Space

- Benign Software vs. Malware
- Binary Code vs. Source Code
- Vulnerability vs. Software Bug
- Memory Error vs. Design Error

Today, we will talk about vulnerability and exploitation analysis of memory errors in the binary code of the benign software.

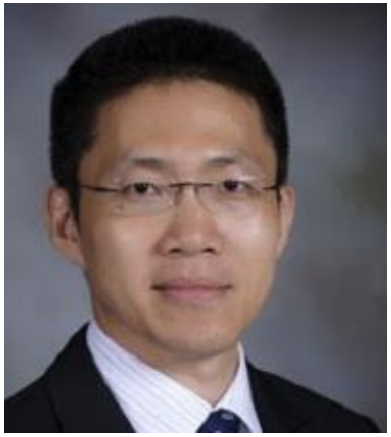
The Speakers



- Lixin (Nathan) Li

  Senior Research Lead
  Cyber Innovation Unit
  Battelle Memorial Institute, USA

Battelle
The Business of Innovation



- Chao Wang

  Assistant Professor
  Dept. Electrical & Computer Engr.
  Virginia Tech, USA

VirginiaTech
Invent the Future

Where are we?

Battelle
The Business of Innovation

- Non-profit, $5.2 billion annually in global R&D
- 22,000 employees in 130 locations worldwide

VirginiaTech
Invent the Future

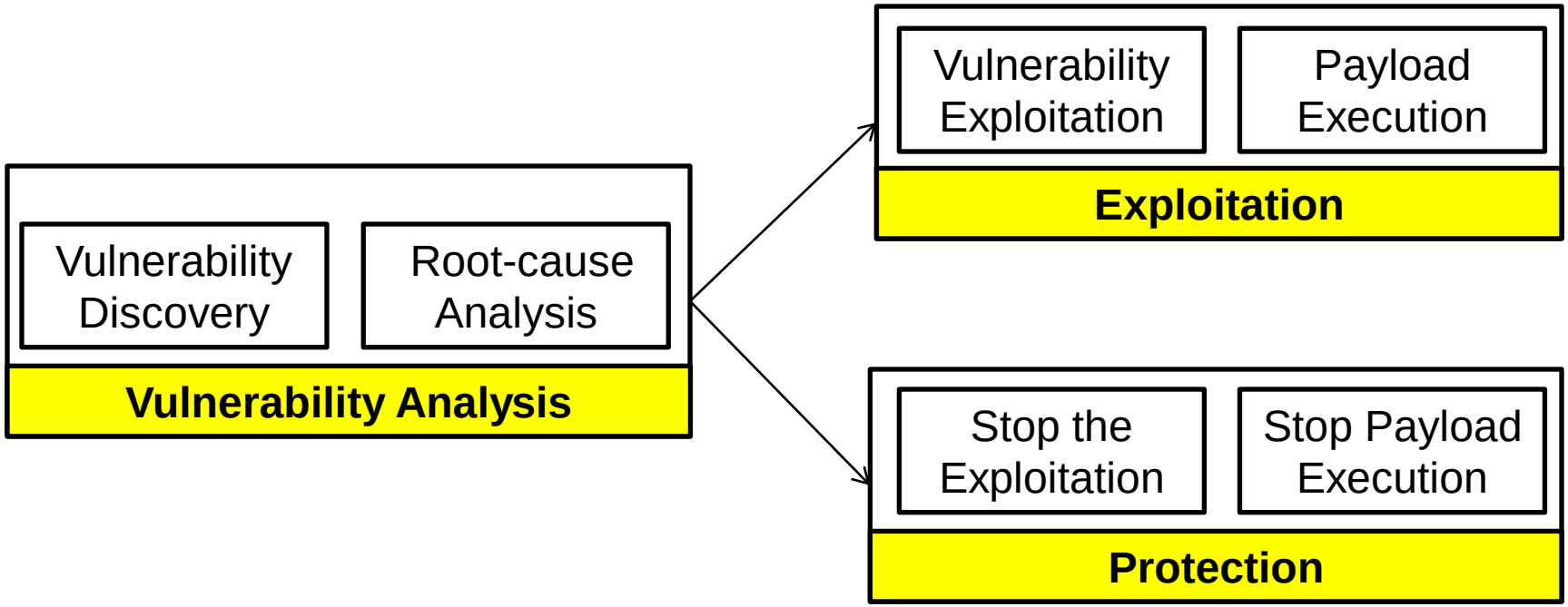
- Land-grant university in Virginia
- 31,087 students (7,228 graduate students)

Outline of today's tutorial

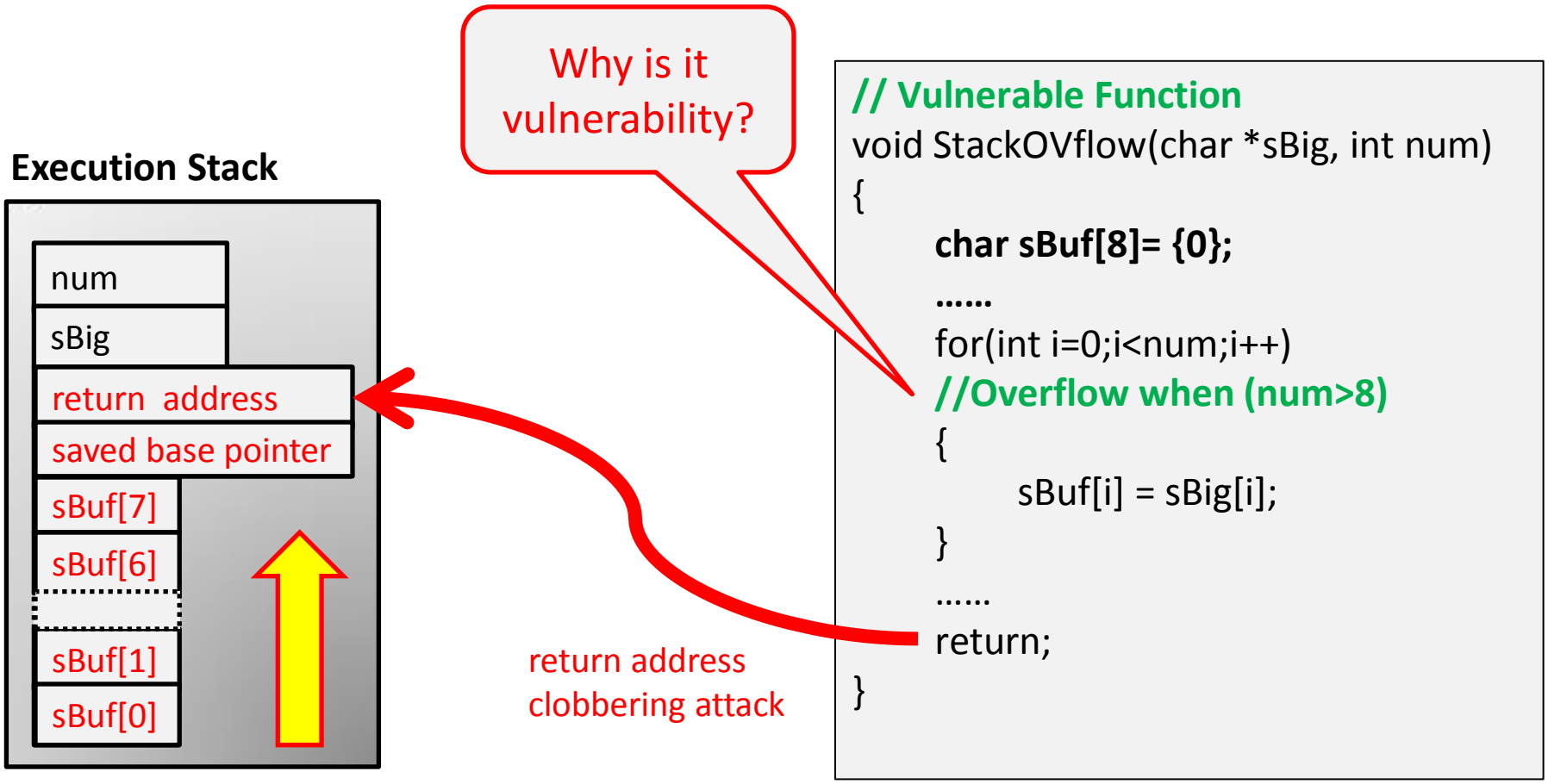
- First Half (90 minutes)
  - **Background**
    - Dynamic analysis and symbolic analysis
    - Their success in vulnerability discovery
    - Challenges in exploitation research
  - Our new security analysis tool
    - Cross-platform trace generation

- Second Half (90 minutes)
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Aspects of Binary Security Analysis



What is a Vulnerability?



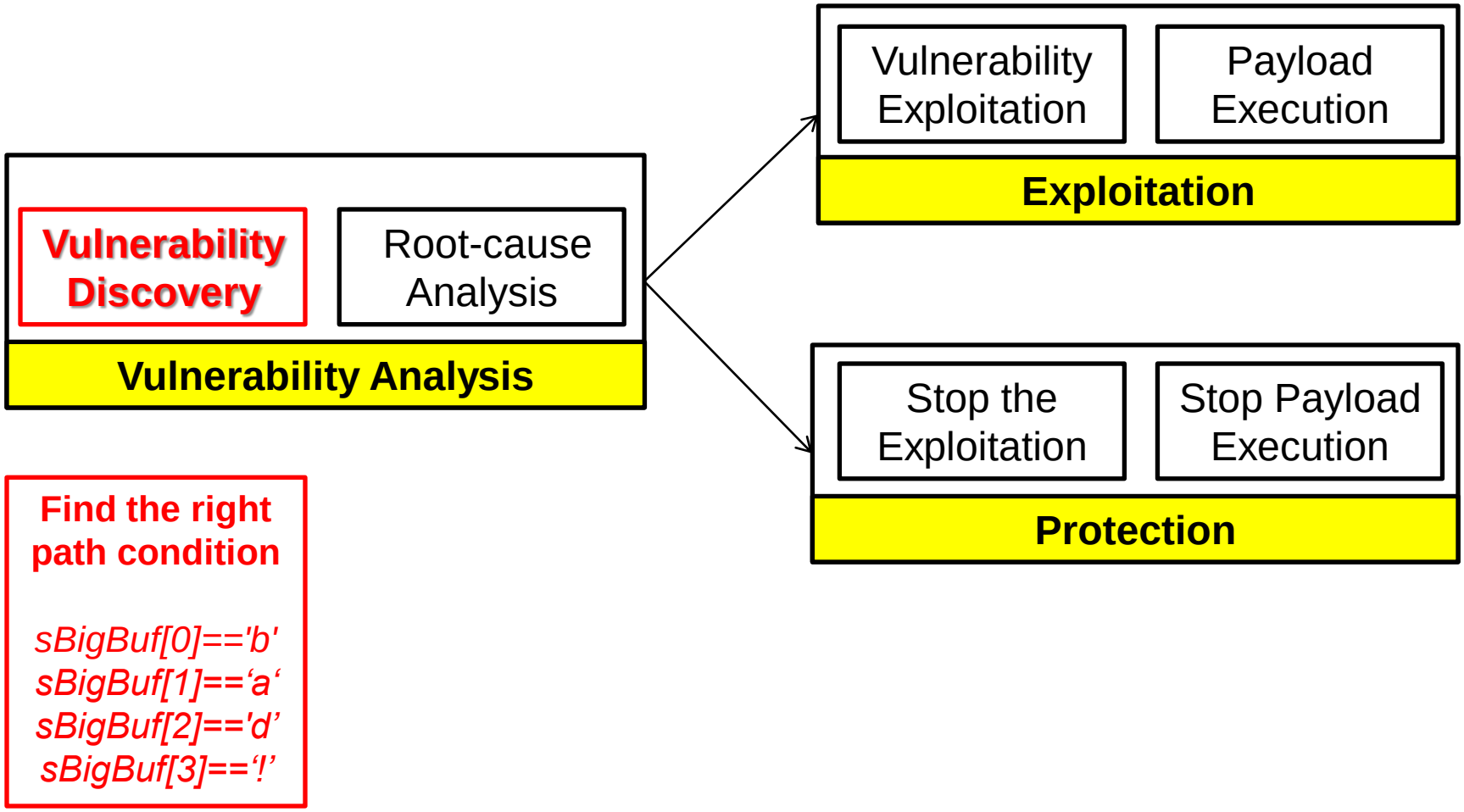
Vulnerability Discovery

```
//INPUT
ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

//INPUT TRANSFORMATIONS
……

//PATH CONDITIONS
if(sBigBuf[0]=='b') iCount++;
if(sBigBuf[1]=='a') iCount++;
if(sBigBuf[2]=='d') iCount++;
if(sBigBuf[3]=='!') iCount++;
if(iCount==4) // bad!
    StackOVflow (sBigBuf, dwBytesRead)
else // Good
    printf("Good!");
```

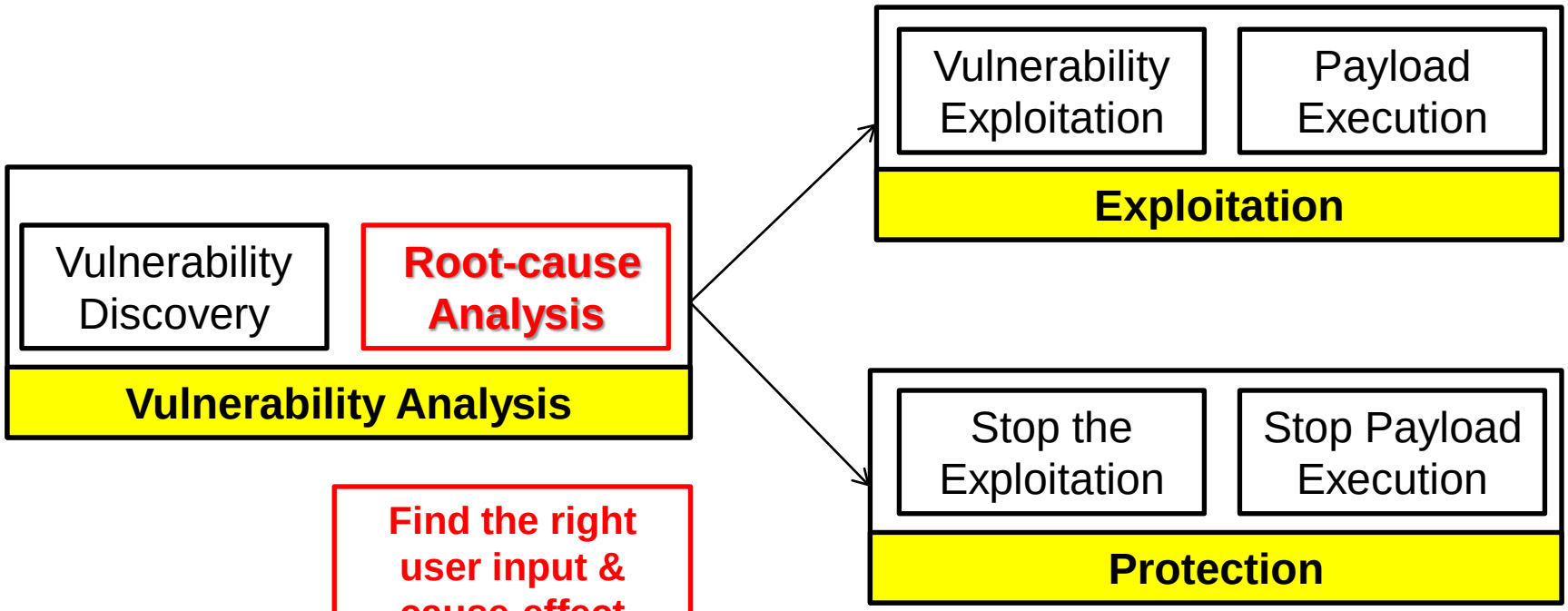
Is this possible?

```
// Vulnerable Function
void StackOVflow(char *sBig,  t num)
{
        char sBuf[8]= {0};
        ……
        for(int i=0;i<num;i++)
        //Overflow when (num>8)
        {
                sBuf[i] = sBig[i];
        }
        ……
        return;
}
```

Phases of Binary Security Analysis



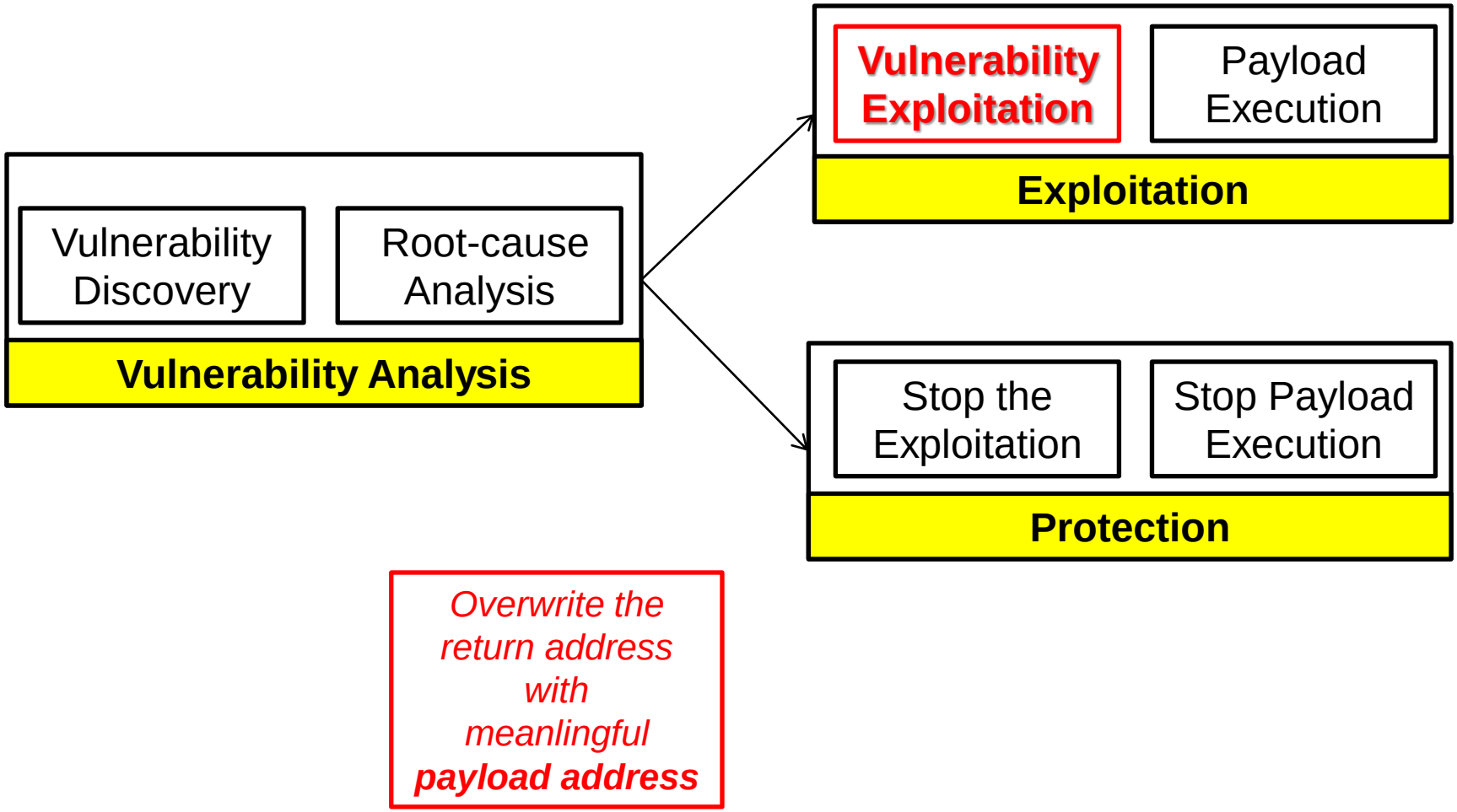
Phases of Binary Security Analysis



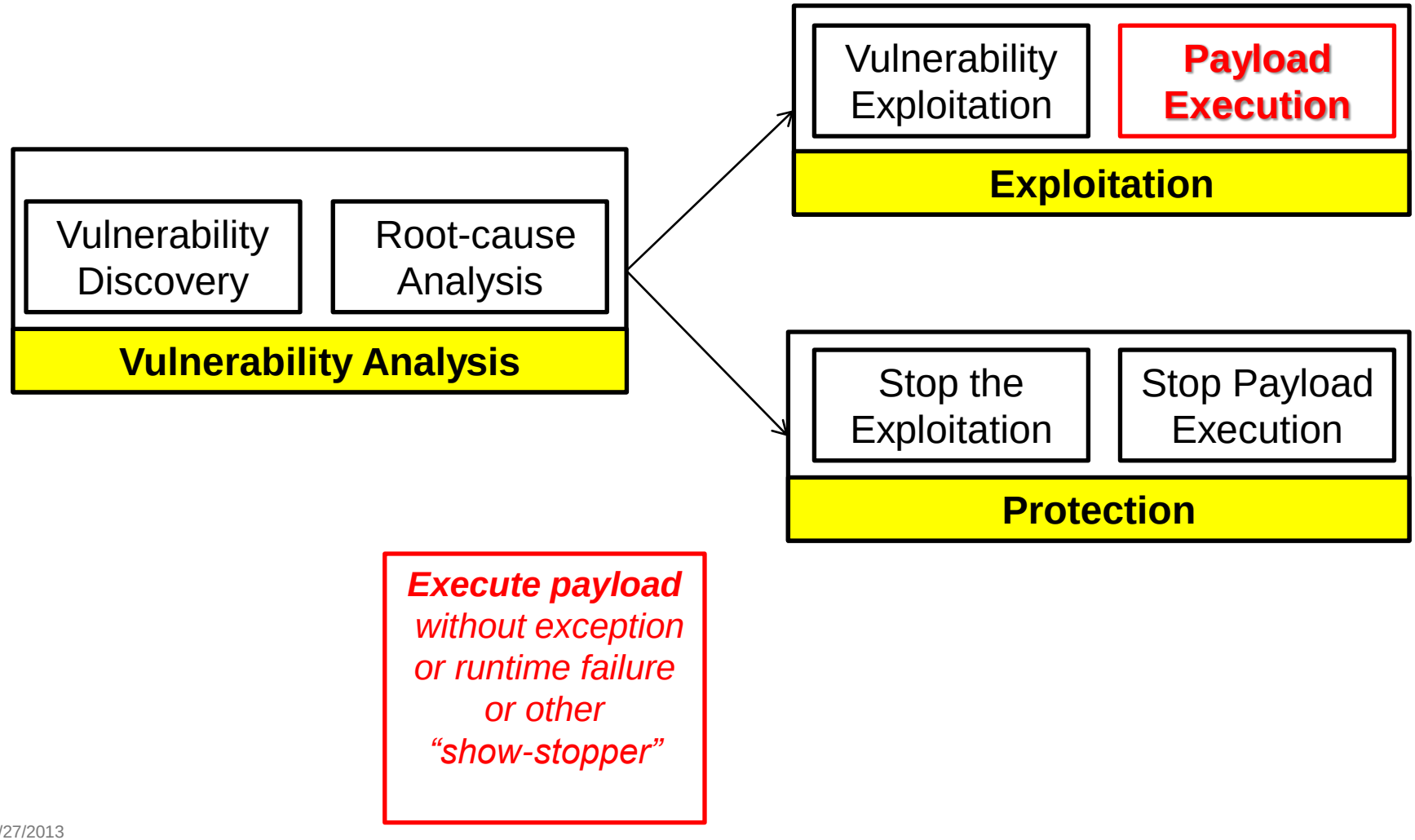
Find the right user input & cause-effect

strlen(sBigBuf) > 8
&
Input[8 -15] overwrite saved stack pointer and return address

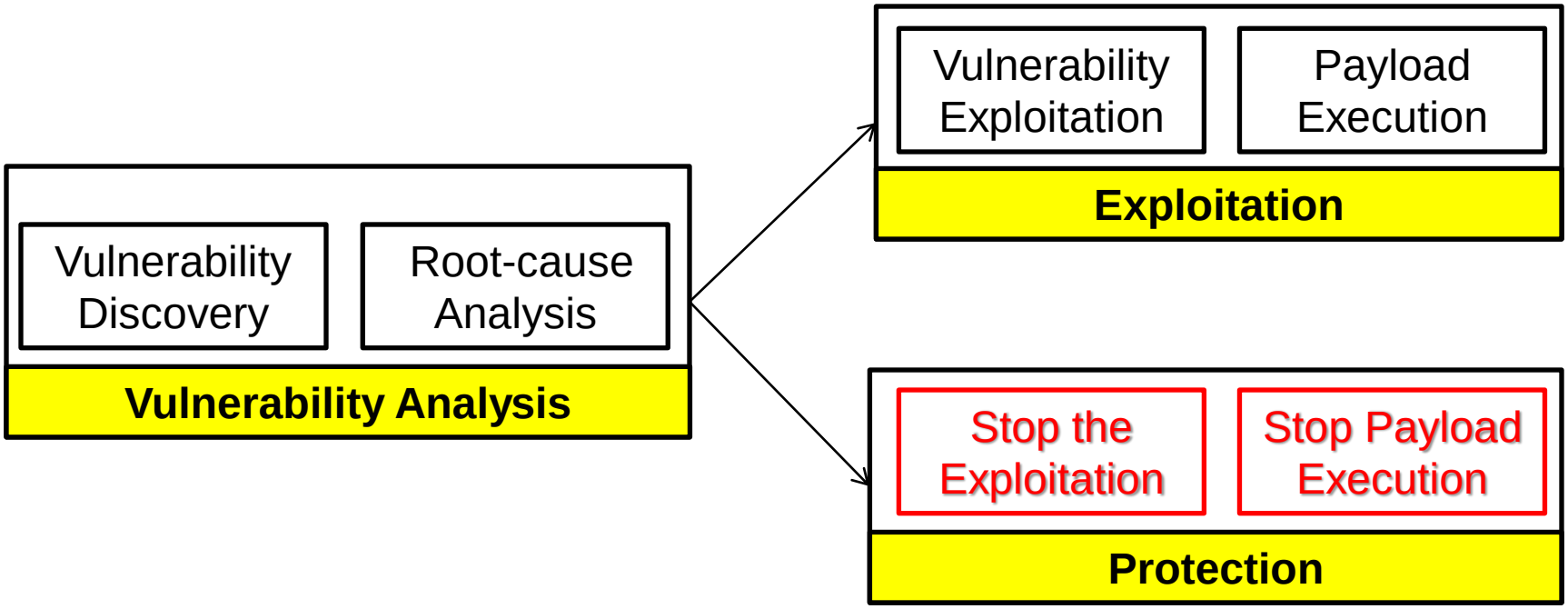
Phases of Binary Security Analysis



Phases of Binary Security Analysis



Phases of Binary Security Analysis

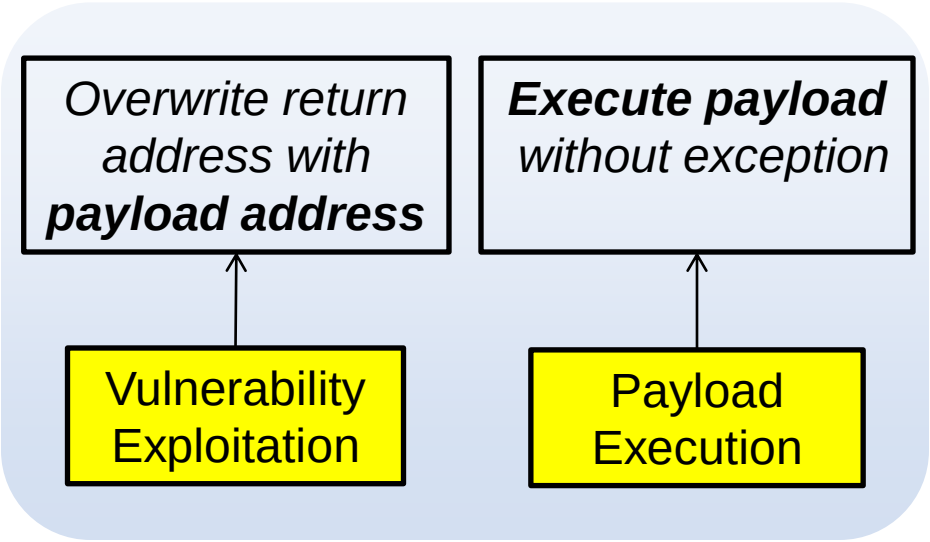


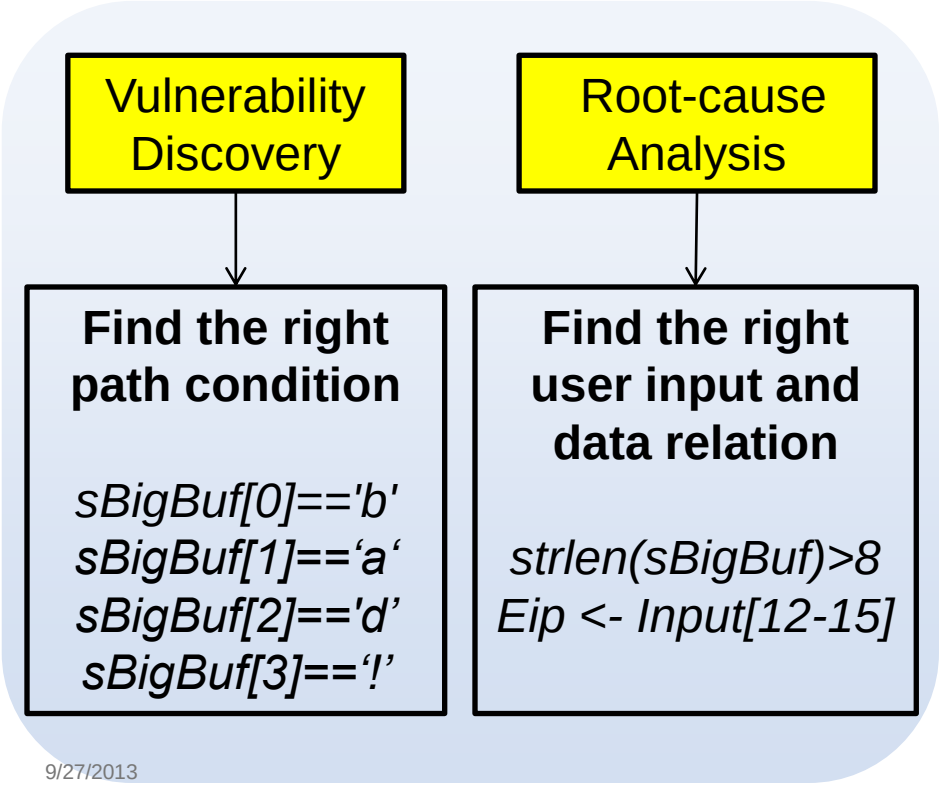
ASLR, DEP, *and* ***protections*** *that make payload address* ***unpredictable*** *& payload* ***un-executable***

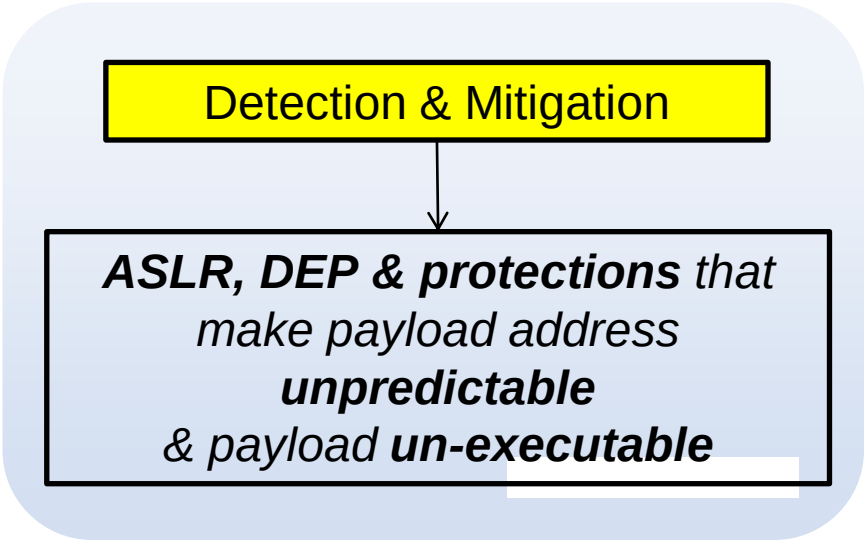
Aspects of Binary Security Analysis

We focus on two methods:

- dynamic taint analysis
- symbolic execution







Outline

- First Half
  - Background
    - **Dynamic taint analysis and symbolic analysis**
    - Their success in vulnerability discovery
    - Challenges in exploitation research
  - Our analysis platform
    - Cross-platform trace generation

- Second Half
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Vulnerability is a property of the program

```
if(iCount==4) // bad!
        StackOVflow(sBigBuf,dwBytesRead)
else // Good
        printf("Good!");
```

```
void StackOVflow(char *sBig,int num)
{
        char sBuf[8]= {0};
        ……
        for(int i=0;i<num;i++)
        //Overflow when num>sizeof(sBuf]
        {
                sBuf[i] = sBig[i];
        }
        ……
        return;
}
```

Dynamic Taint Analysis

Program

X := 8 + get_input()

Y := X + 2

Goto Y

Concrete Value

X → 0x12345678

Y → 0x1234567A

Use (Y) as address

Is Tainted?

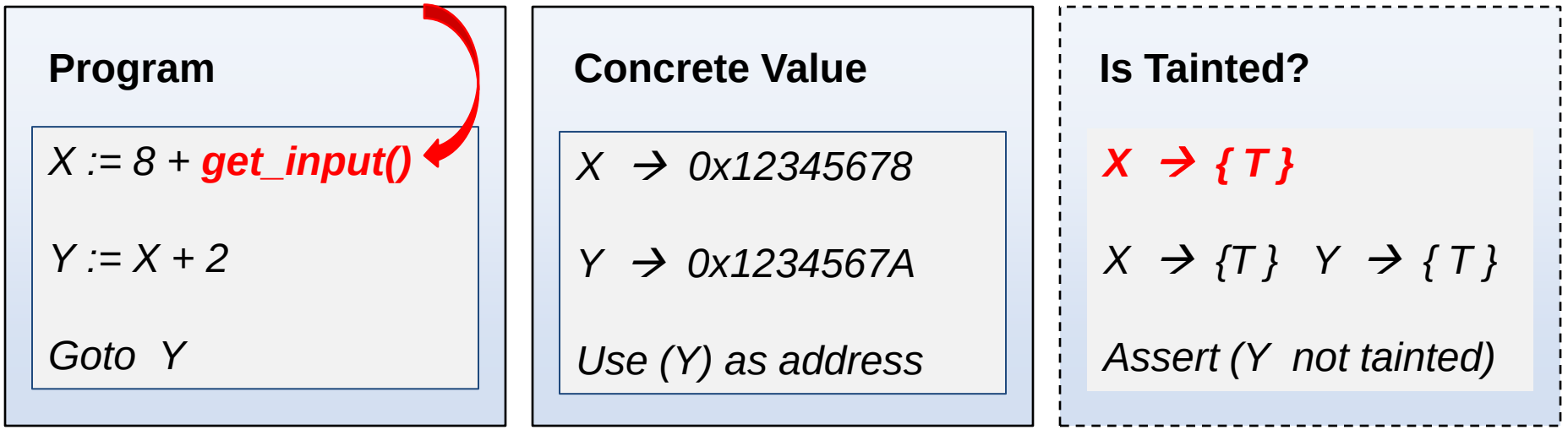
X → { T }

X → {T } Y → { T }

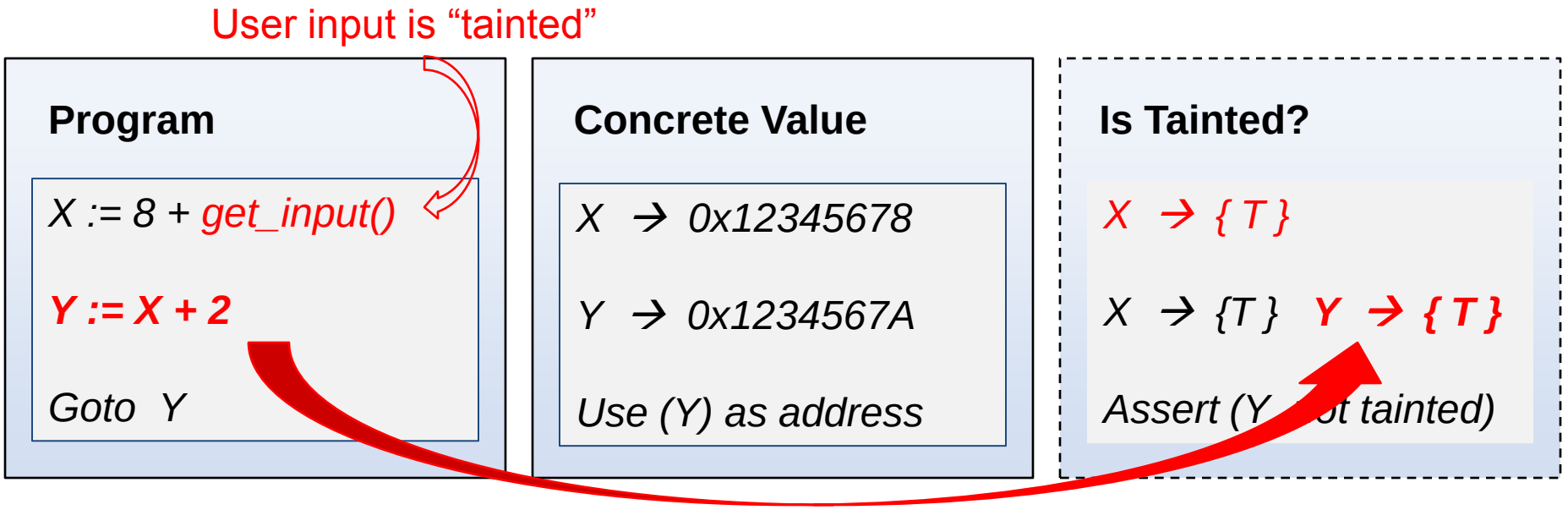
Assert (Y not tainted)

Dynamic Taint Analysis

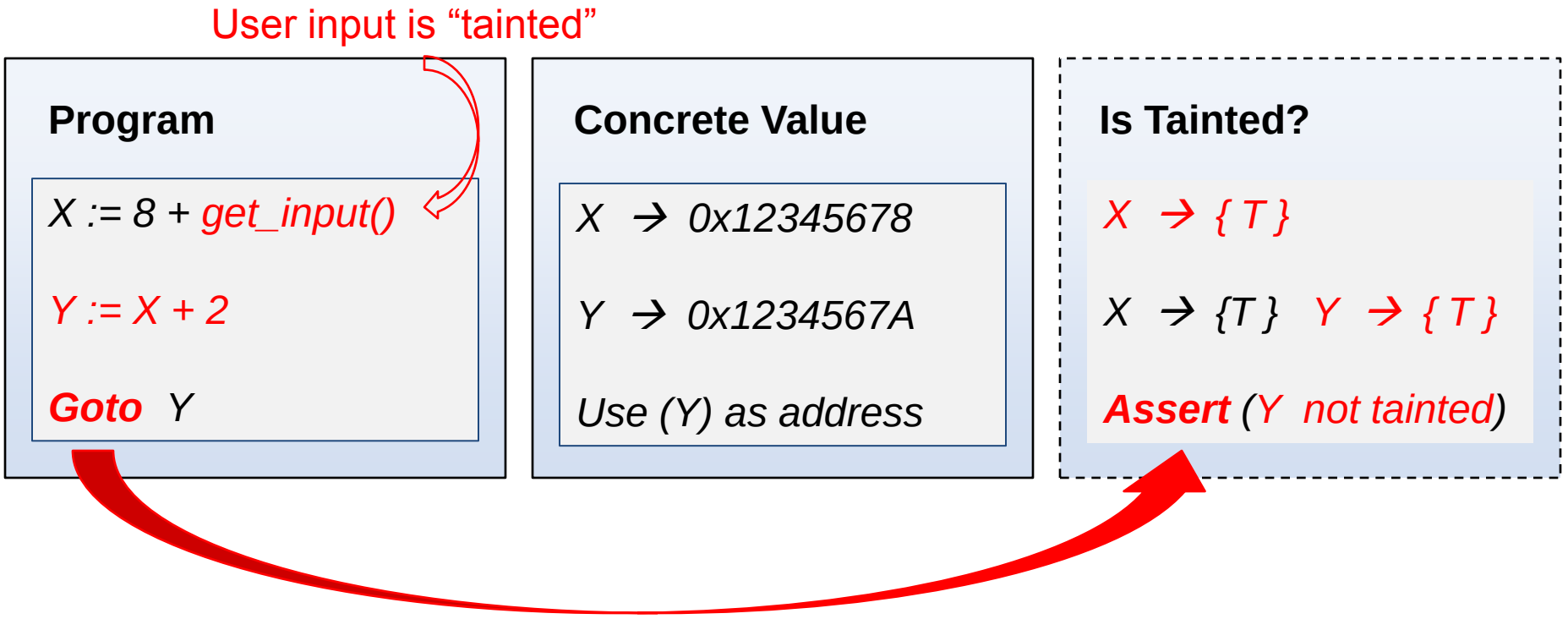
User input is "tainted"



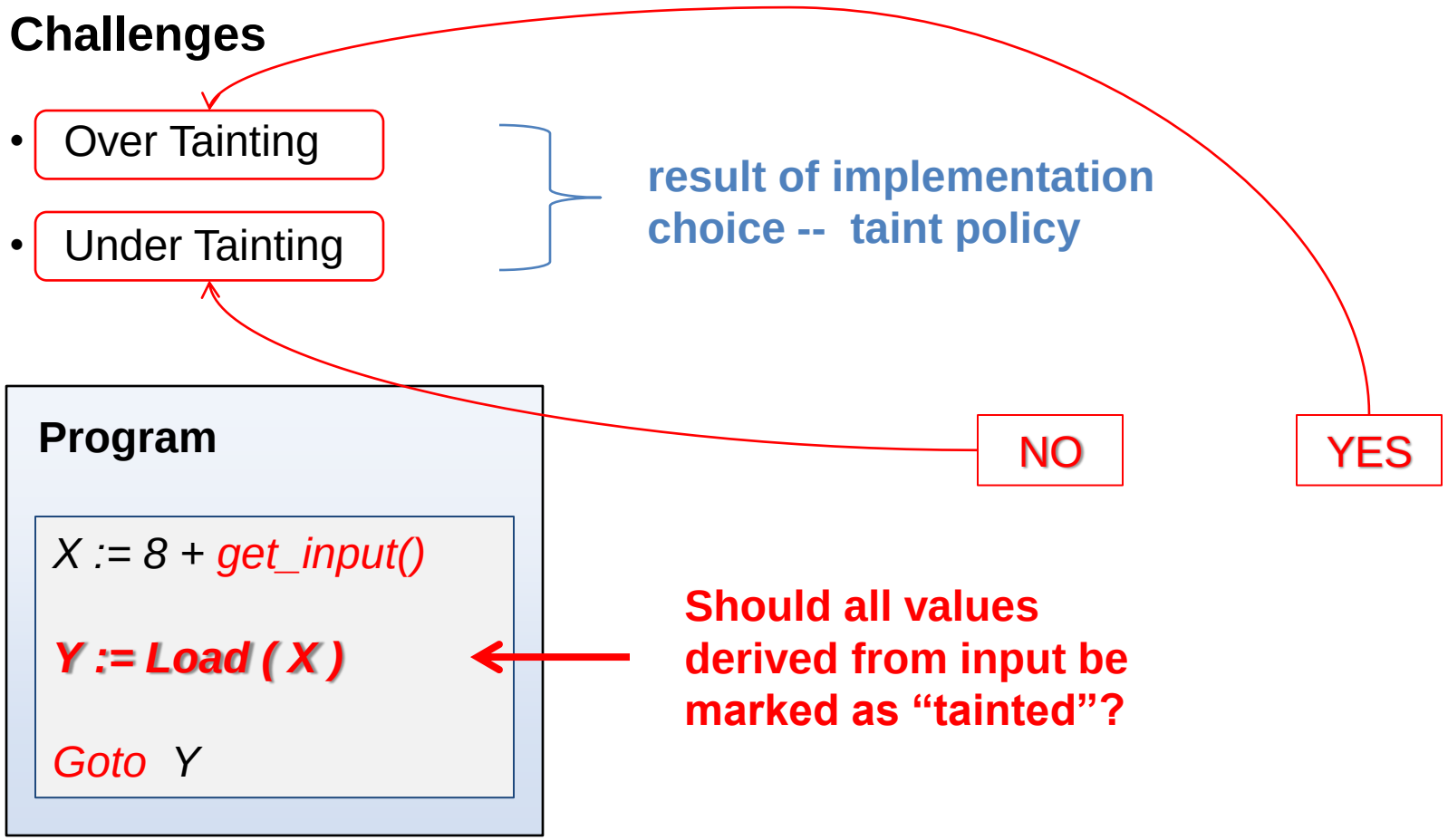
Dynamic Taint Analysis



Dynamic Taint Analysis



Dynamic Taint Analysis



Dynamic Taint Analysis

Challenges

- Over Tainting
- Under Tainting

```
num := get_input()
…
sBuf[i] := sBig[i]

… (one million instructions)

Goto  return address;
```

Outline

- First Half
  - Background
    - Dynamic taint analysis and **symbolic analysis**
    - Their success in vulnerability discovery
    - Challenges in exploitation research
  - Our analysis platform
    - Cross-platform trace generation

- Second Half
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Symbolic Execution

Program

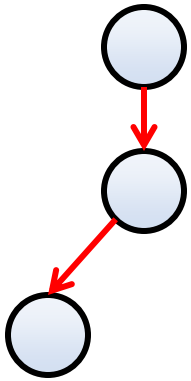
```
X := 8 + get_input()

If ( X – 10 == 5)
    Goto L2;
Else
    Goto L3;
```

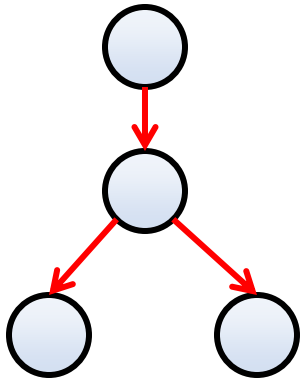
Concrete Execution

```
X → 15

(15 – 10 == 5)
    Goto L2;
```

Symbolic Execution

```
X → 8 + s

(8+s-10==5) → True
    Goto L2;
(8+s-10!=5) → False
    Goto L3;
```





Symbolic Execution

(path-cond, loc)

Current location and path condition

mem-map

Current content of the symbolic memory

Program

```
X := 8 + get_input()

If ( X – 10 == 5)
    Goto L2;
Else
    Goto L3;
```

SymbolicExec(Program P) {
  init_state ← $\langle true, l_{init}, \mathcal{M} \rangle$;
  stack.push(init_state);
  while (stack is not empty) {
    $\langle pcon, l \rangle$ ← stack.pop();
    if ($pcon$ is satisfiable) {
      for each (event $l \xrightarrow{instr} l'$ at location l) {
        if ($instr$ is **abort**)
          return ∅; //BUG_FOUND;
        else if ($instr$ is **halt**)
          τ ← solve ($pcon$);
          $\mathcal{T} := \mathcal{T} \cup \{\tau\}$;
        else if ($instr$ is $if(c)$)
          next_state ← $\langle pcon \wedge c, l', \mathcal{M} \rangle$;
          stack.push(next_state);
        else if ($instr$ is $v := exp$)
          $\mathcal{M}'$ ← update($\mathcal{M}, v, exp$);
          next_state ← $\langle pcon, l', \mathcal{M}' \rangle$;
          stack.push(next_state);
      }
    }
  }
  return $\mathcal{T}$;
}

Symbolic Execution

(path-cond, loc)

(true, L0)

mem-map

x → *undef*

Program

L0 →

X := 8 + get_input()

If (X – 10 == 5)
Goto L2;
Else
Goto L3;

SymbolicExec(Program P) {
  init_state ← $\langle true, l_{init}, \mathcal{M} \rangle$;
  stack.push(init_state);
  while (stack is not empty) {
    $\langle pcon, l \rangle$ ← stack.pop();
    if ($pcon$ is satisfiable) {
      for each (event $l \xrightarrow{instr} l'$ at location l) {
        if ($instr$ is **abort**)
          return ∅; //BUG_FOUND;
        else if ($instr$ is **halt**)
          τ ← solve ($pcon$);
          $\mathcal{T} := \mathcal{T} \cup \{\tau\}$;
        else if ($instr$ is $if(c)$)
          next_state ← $\langle pcon \wedge c, l', \mathcal{M} \rangle$;
          stack.push(next_state);
        else if ($instr$ is $v := exp$)
          $\mathcal{M}'$ ← update($\mathcal{M}, v, exp$);
          next_state ← $\langle pcon, l', \mathcal{M}' \rangle$;
          stack.push(next_state);
      }
    }
  }
  return $\mathcal{T}$;
}

Symbolic Execution

(path-cond, loc)

(true, L0)
(true, L1)

mem-map

X → *(8+s)*

Program

L1 →

```
X := 8 + get_input()

If ( X – 10 == 5)
    Goto L2;
Else
    Goto L3;
```

```
SymbolicExec( Program P ) {
    init_state ← ⟨true, l_init, M⟩;
    stack.push( init_state );
    while ( stack is not empty ) {
        ⟨pcon, l⟩ ← stack.pop();
        if ( pcon is satisfiable ) {
            for each ( event l —instr→ l' at location l ) {
                if ( instr is abort )
                    return ∅;  //BUG_FOUND;
                else if ( instr is halt )
                    τ ← solve (pcon);
                    T := T ∪ {τ};
                else if ( instr is if(c) )
                    next_state ← ⟨pcon ∧ c, l', M⟩;
                    stack.push( next_state );
                else if ( instr is v := exp )
                    M' ← update(M, v, exp);
                    next_state ← ⟨pcon, l', M'⟩;
                    stack.push( next_state );
            }
        }
    }
    return T;
}
```

Symbolic Execution

(path-cond, loc)

(true, L0)
(*(8+s-10==5)*, L2)
(*(8+s-10 !=5)*, L3)

mem-map

x → *(8+s)*

Program

X := 8 + get_input()

If (X – 10 == 5)
    Goto L2;
Else
    Goto L3;

L2 →

L3 →

```
SymbolicExec( Program P ) {
    init_state ← ⟨true, l_init, M⟩;
    stack.push( init_state );
    while ( stack is not empty ) {
        ⟨pcon, l⟩ ← stack.pop();
        if ( pcon is satisfiable ) {
            for each ( event l --instr--> l' at location l ) {
                if ( instr is abort )
                    return ∅;  //BUG_FOUND;
                else if ( instr is halt )
                    τ ← solve (pcon);
                    T := T ∪ {τ};
                else if ( instr is if(c) )
                    next_state ← ⟨pcon ∧ c, l', M⟩;
                    stack.push( next_state );
                else if ( instr is v := exp )
                    M' ← update(M, v, exp);
                    next_state ← ⟨pcon, l', M'⟩;
                    stack.push( next_state );
            }
        }
    }
    return T;
}
```

Symbolic Execution

(path-cond, loc)

(true, L0)
((8+s-10==5), L2)
(*(8+s-10 !=5)*, L3)

mem-map

x → *(8+s)*

Program

```
X := 8 + get_input()

If ( X – 10 == 5)
    Goto L2;
Else
    Goto L3;
```

L2 →

L3 →

What input can lead to L2?

```
SymbolicExec( Program P ) {
    init_state ← ⟨true, l_init, M⟩;
    stack.push( init_state );
    while ( stack is not empty ) {
        ⟨pcon, l⟩ ← stack.pop();
        if ( pcon is satisfiable ) {
            for each ( event l --instr--> l' at location l ) {
                if ( instr is abort )
                    return ∅;  //BUG_FOUND;
                else if ( instr is halt )
                    τ ← solve (pcon);
                    T := T ∪ {τ};
                else if ( instr is if(c) )
                    next_state ← ⟨pcon ∧ c, l', M⟩;
                    stack.push( next_state );
                else if ( instr is v := exp )
                    M' ← update(M, v, exp);
                    next_state ← ⟨pcon, l', M'⟩;
                    stack.push( next_state );
            }
        }
    }
    return T;
}
```

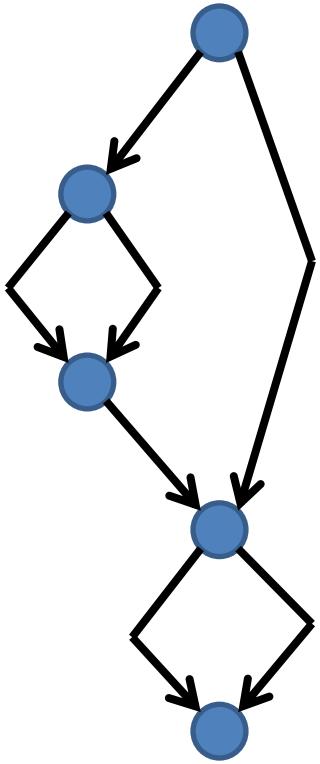
Symbolic Execution

3 branches can lead to 8 paths

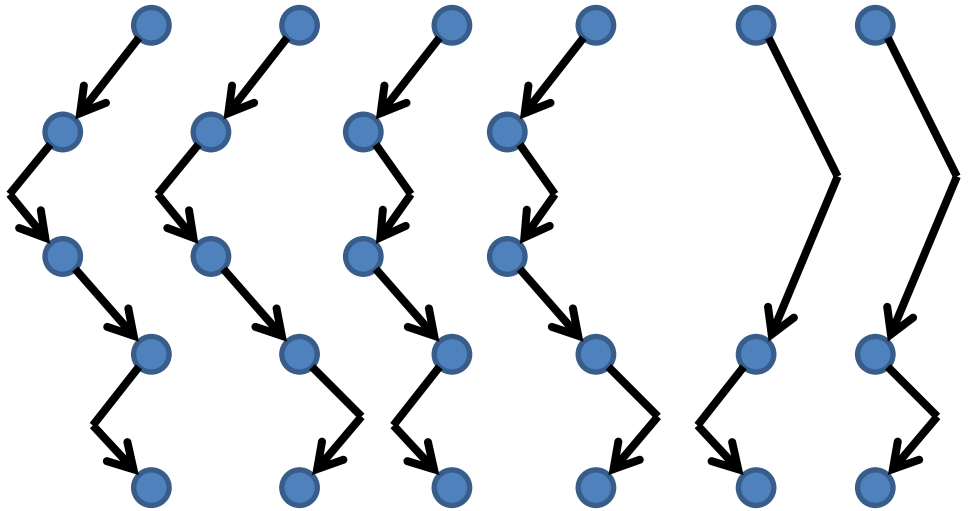
```
1:  if (a<=0) res = res+1;
2:  else      res = res-1;
    ...
3:  if (b<=0) res = res+2;
4:  else      res = res-2;
    ...
5:  if (c<=0) res = res+3;
6:  else      res = res-3;
```

```
P1  P2  P3  P4  P5  P6  P7  P8
-----------------------------
1   1   1   1   2   2   2   2
|   |   |   |   |   |   |   |
3   3   4   4   3   3   4   4
|   |   |   |   |   |   |   |
5   6   5   6   5   6   5   6
|   |   |   |   |   |   |   |
```

Symbolic Execution



Path explosion: the number of paths can be exponential in the number of branches



Challenges in symbolic execution

- Path Explosion → exploration strategy
- Symbolic Memory
- System/Library Calls

Over Approximation or **Under Approximation**

Over Approximation → Y can have any value

Under Approximation → Y has a concrete value

Program

X := get_input()

Y := ***Load (X)***

Program

X := get_input()

Y := ***Sqrt (X)***

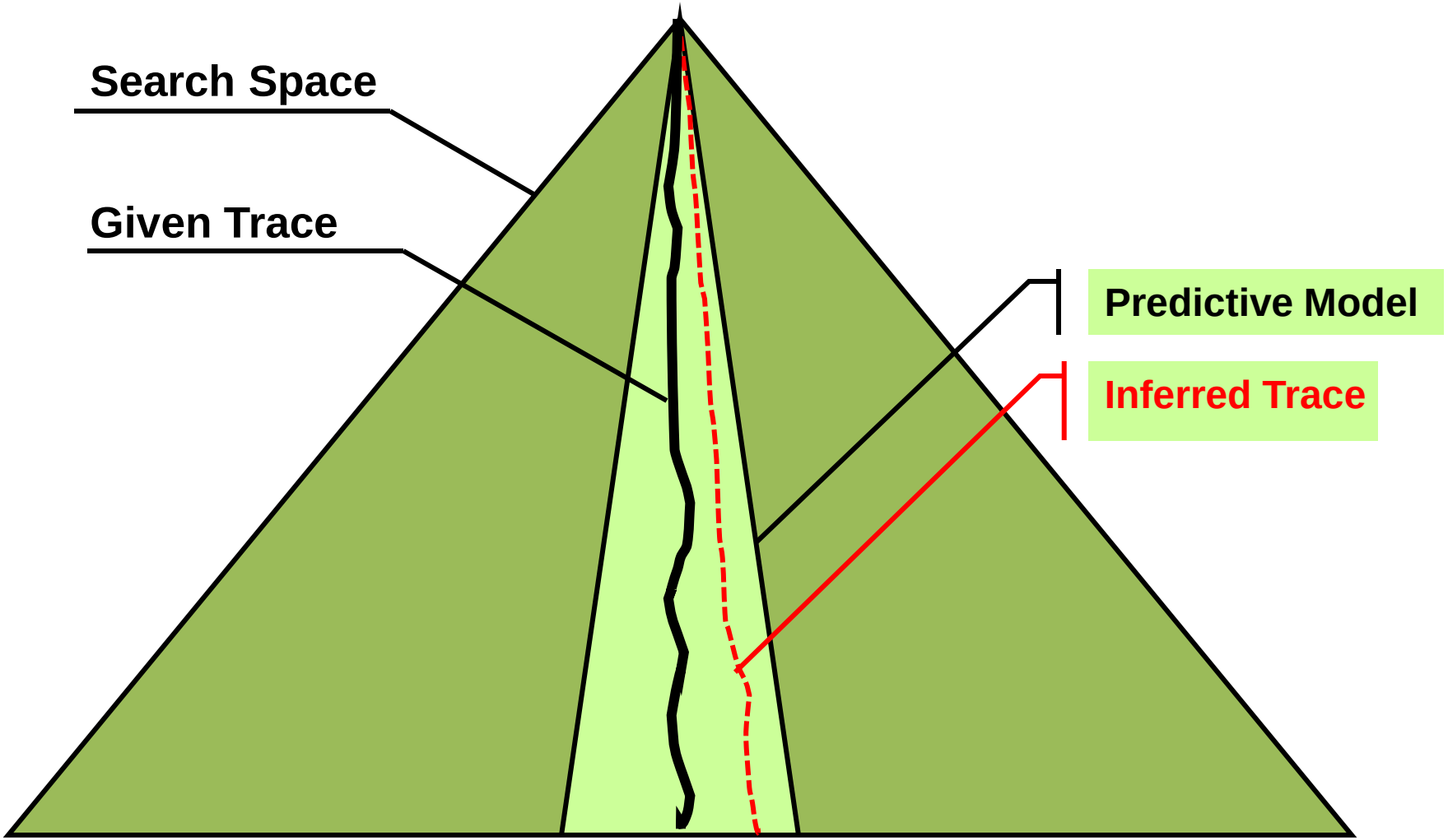
Challenge in Multi-core Systems

- Dynamic taint analysis
  - must model the *thread interference*
- Symbolic execution
  - must deal with the *thread interleaving explosion,* in addition to the path explosion within each thread

Symbolic Predictive Analysis

- Statically analyzing a logged execution trace of a multithreaded program without re-running the program…

Symbolic Predictive Analysis (Rosu, Sen, et al.)



Predicting Vulnerabilities: How?

- Given run is good, but …

Thread1

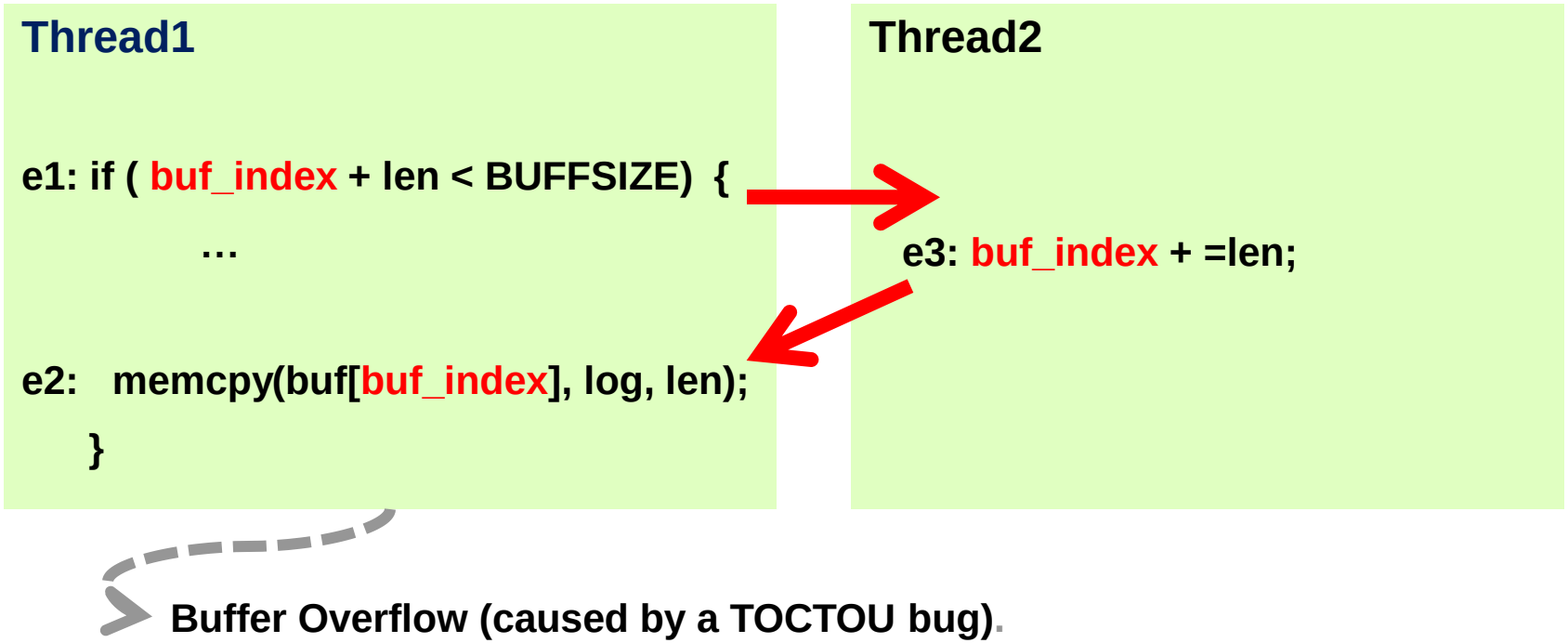
```
e1: if ( buf_index + len < BUFFSIZE)  {
                …
e2:    memcpy(buf[buf_index], log, len);
      }
```

Thread2

```
e3: buf_index + =len;
```

Predicting Errors: How?

- Given run is good, but an alternative run is buggy



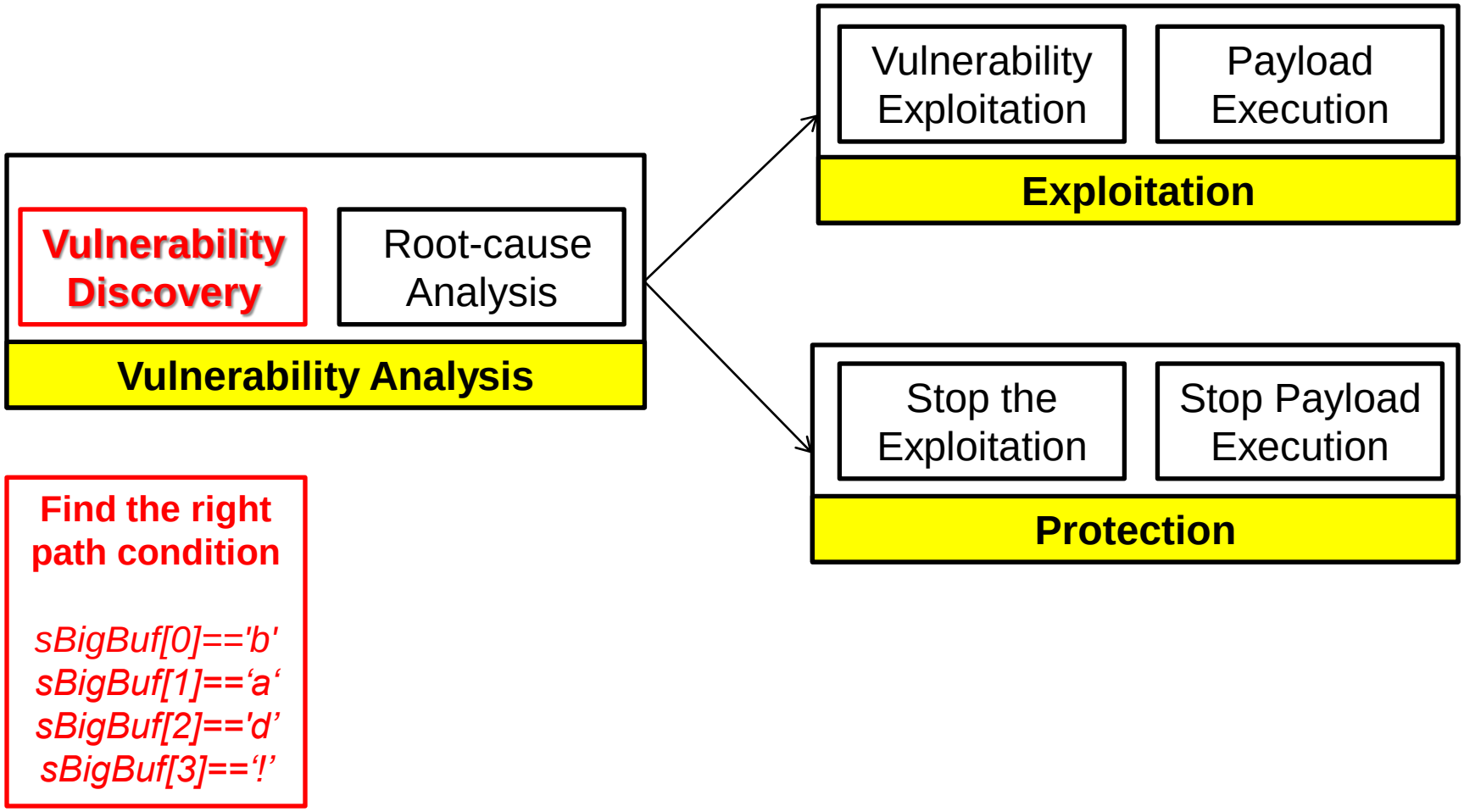
RV 2011: **Predicting Concurrency Failures in Generalized Traces of x86 Executable**, by Chao Wang and Malay Ganai

Outline

- First Half
  - Background
    - Dynamic analysis and symbolic analysis
    - **Success in vulnerability discovery**
    - Challenges in exploitation analysis
  - Our analysis platform
    - Cross-platform trace generation

- Second Half
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Phases of Binary Security Analysis



How are vulnerabilities discovered in practice?

- **Black-box** (Random) **Fuzzing**

  "*Fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer… The field of fuzzing originates with Barton Miller at the University of Wisconsin in 1988*"

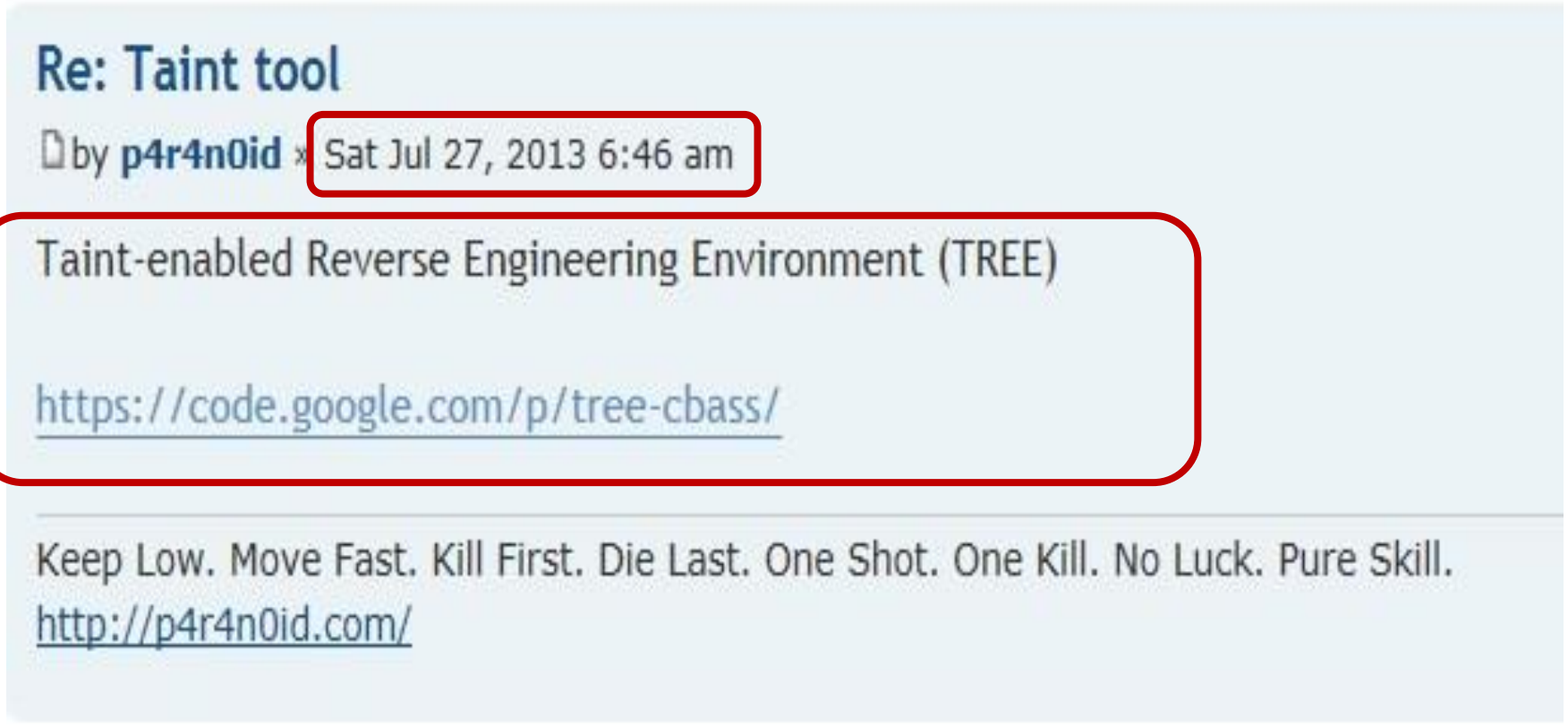
  - Simple, easy to start, effective for low-hanging fruit
  - Poor code coverage

- **Recently, white-box** (symbolic execution based) **Fuzzing**
  - *DART, CUTE, SAGE, KLEE, Cloud9, etc.*

How are vulnerabilities discovered in practice?

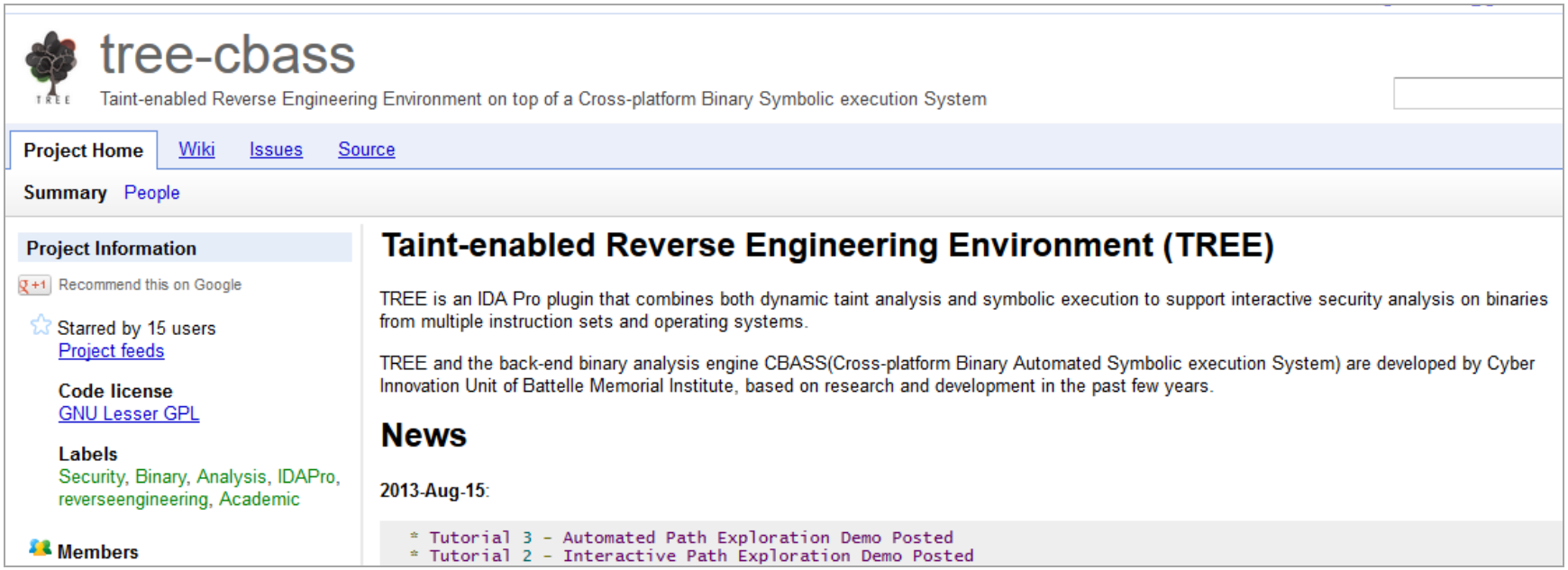
- **Dynamic taint analysis**
  - *BitBlaze*
  - *Argos [EuroSys 2006]*
  - *DynTan [ISSTA 2007]*
  - *DTAM [FSE 2012] for multithreaded programs, etc.*

Our New Tool: Taint-enabled Reverse Engineering Environment (TREE)



Our New Tool: Taint-enabled Reverse Engineering Environment (TREE)

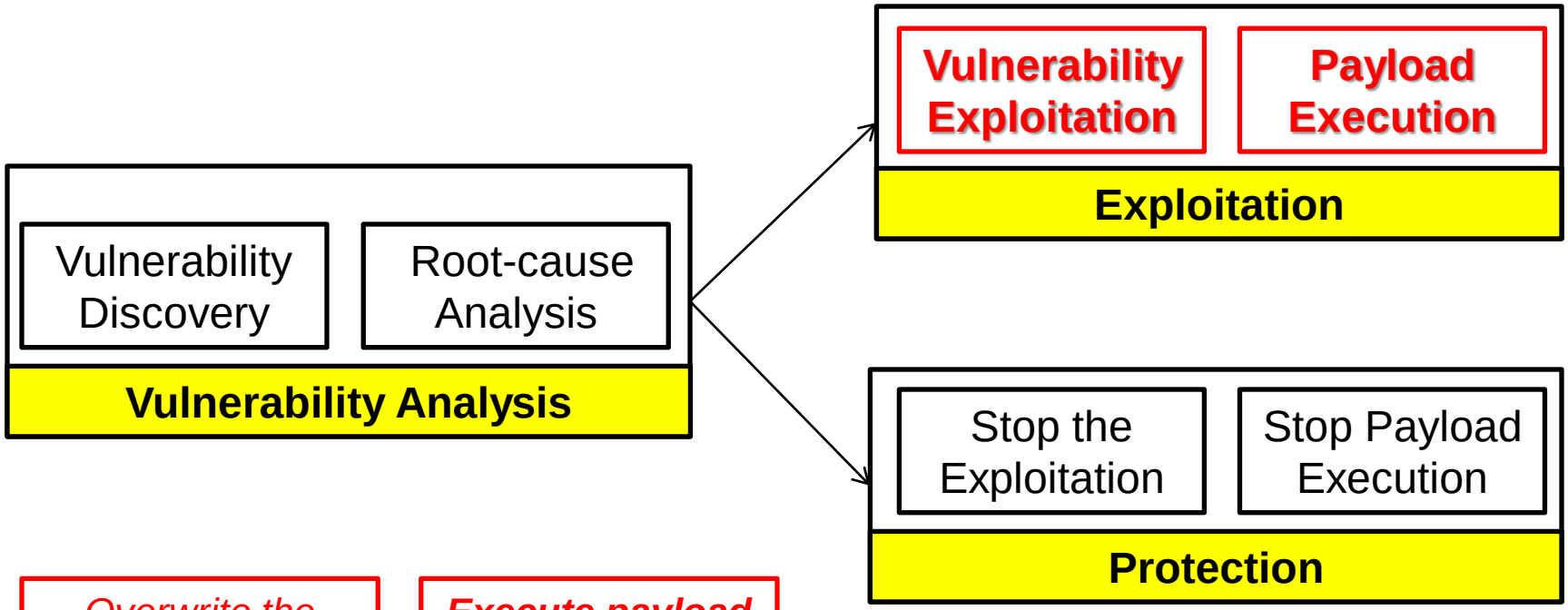
http://code.google.com/p/tree-cbass/



Outline

- First Half
  - Background
    - Dynamic analysis and symbolic analysis
    - Their success in vulnerability discovery
    - **Challenges in exploitation analysis**
  - Our new security analysis platform
    - Cross-platform trace generation
- Second Half
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Exploitation Analysis



Overwrite the return address with meanlingful ***payload address***

Execute payload *without exception or runtime failure or other "show-stopper"*

Why is exploitation hard?

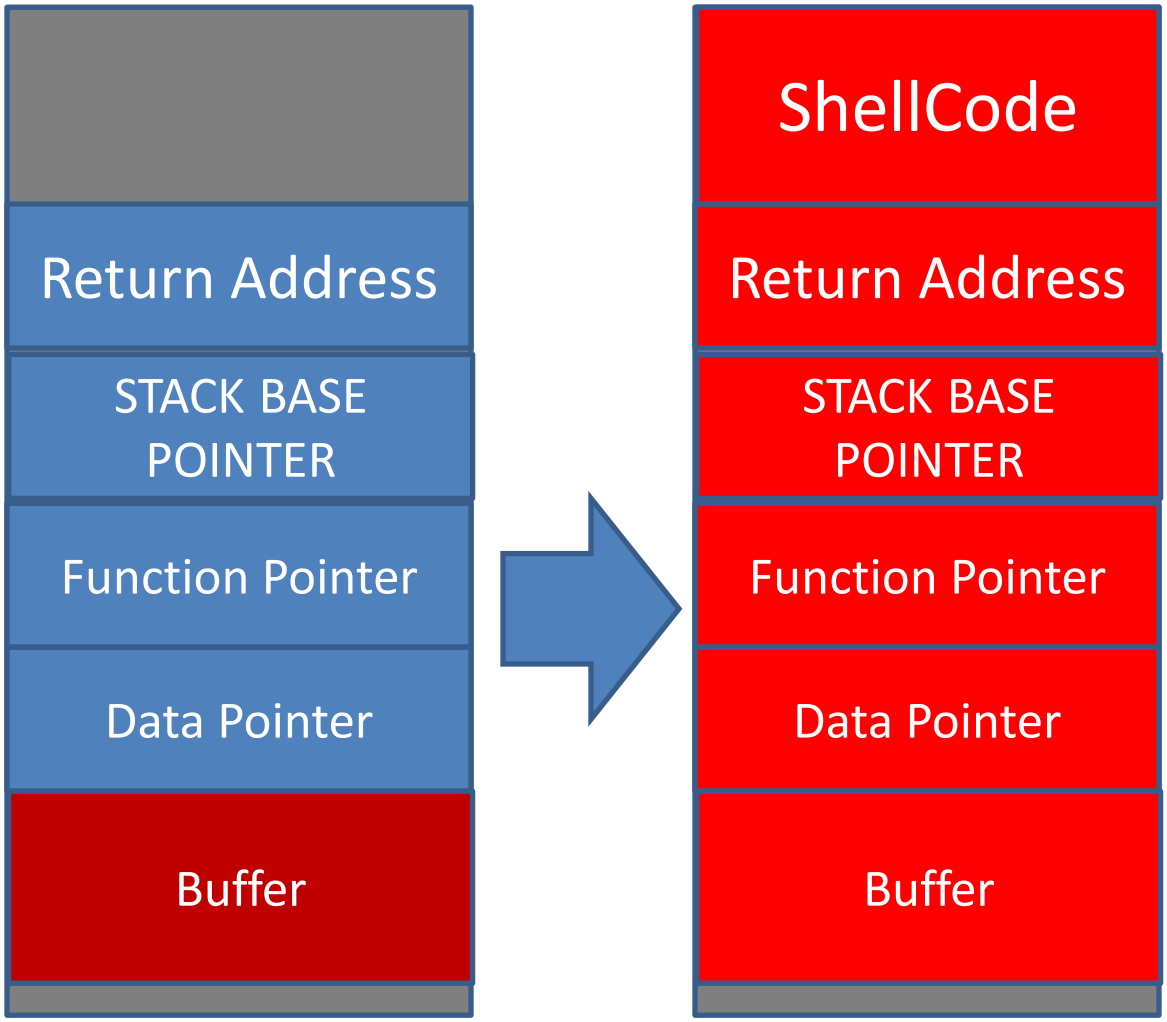
- **Operating system has protection mechanism**
  - **Canary**

  - **NX** (No-Execute)

  - **ASLR** (Address Space Layout Randomization)

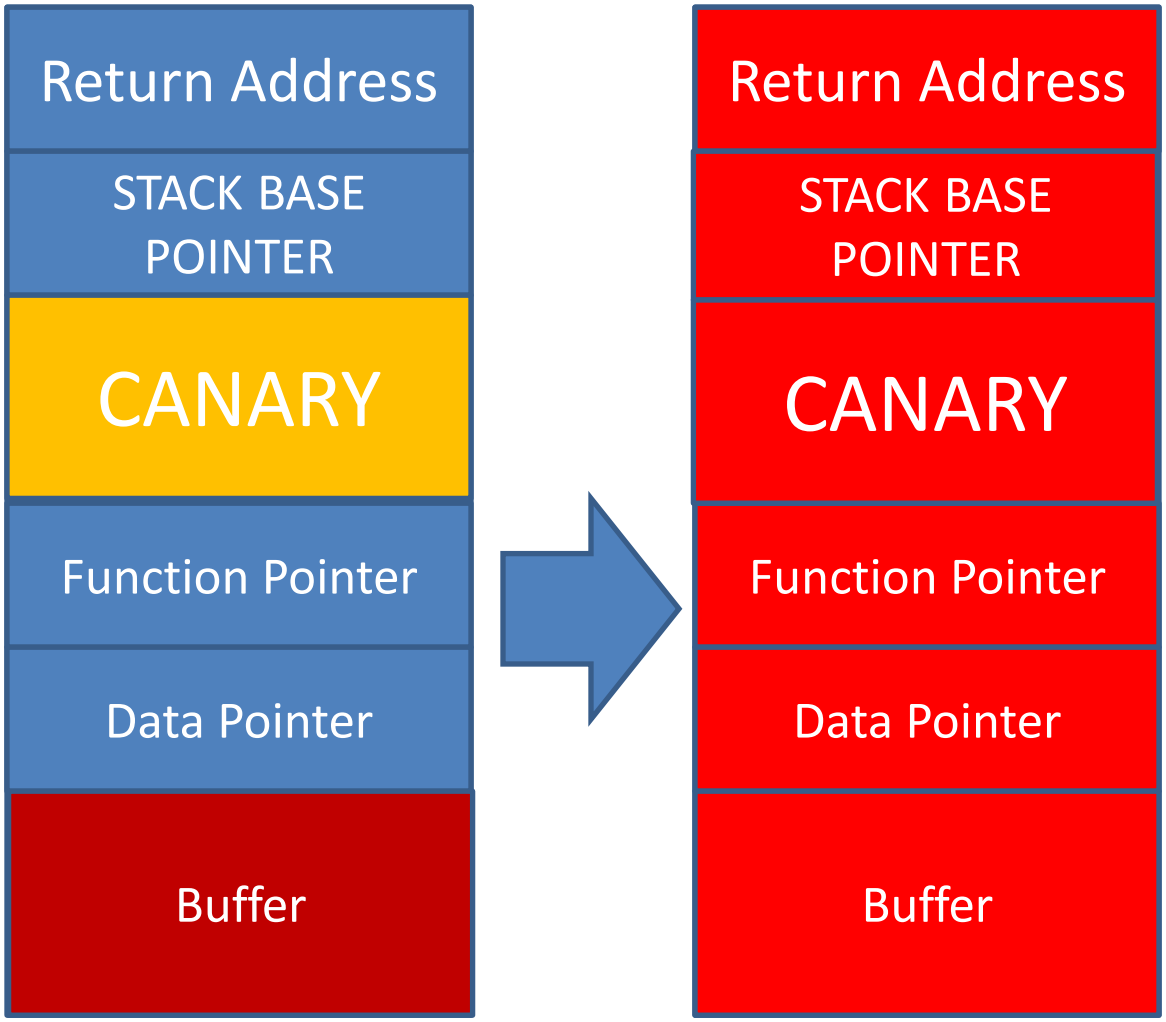
In the old days…

- Exploitation was easy
  - *Clobber return address on the execution stack*



When exploitation hits a “canary”

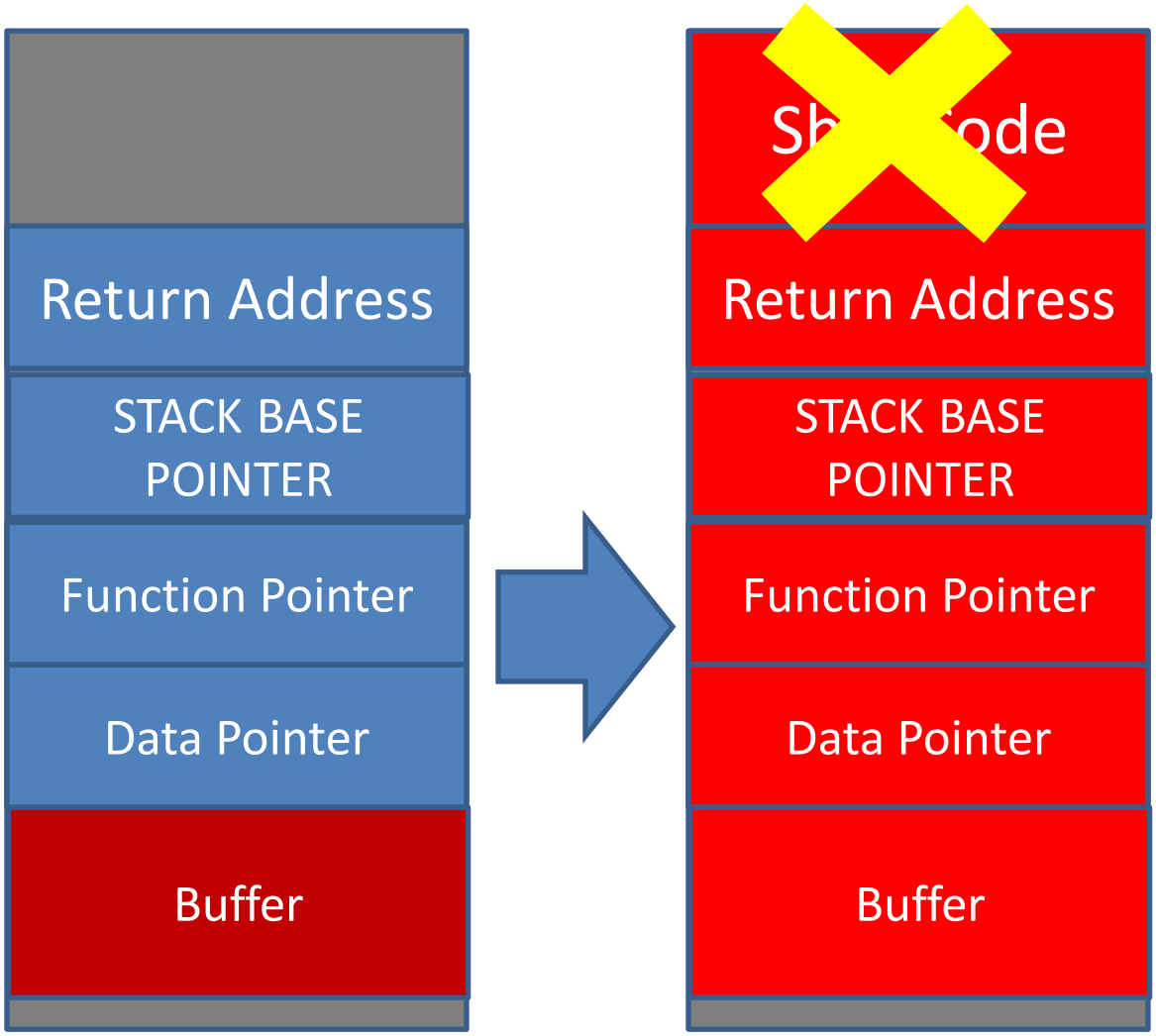
- Stack canary:
  - *Protection added at compiler time*
  - *protect return address and stack base pointer.*



When exploitation hits “NX”

NX: No-eXcute

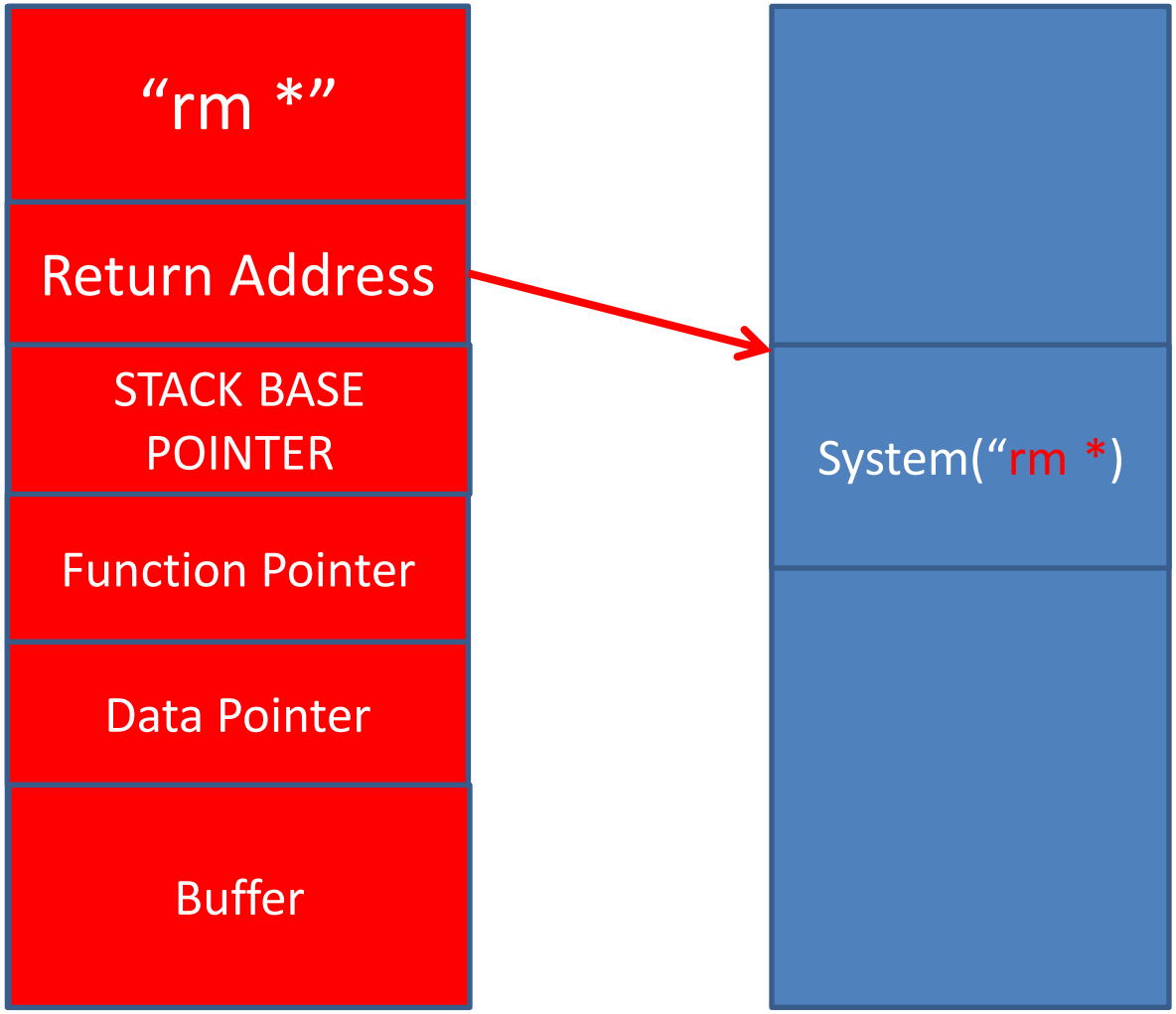
- Block Code Injection Attacks



When exploitation hits “NX”

Defeat NX

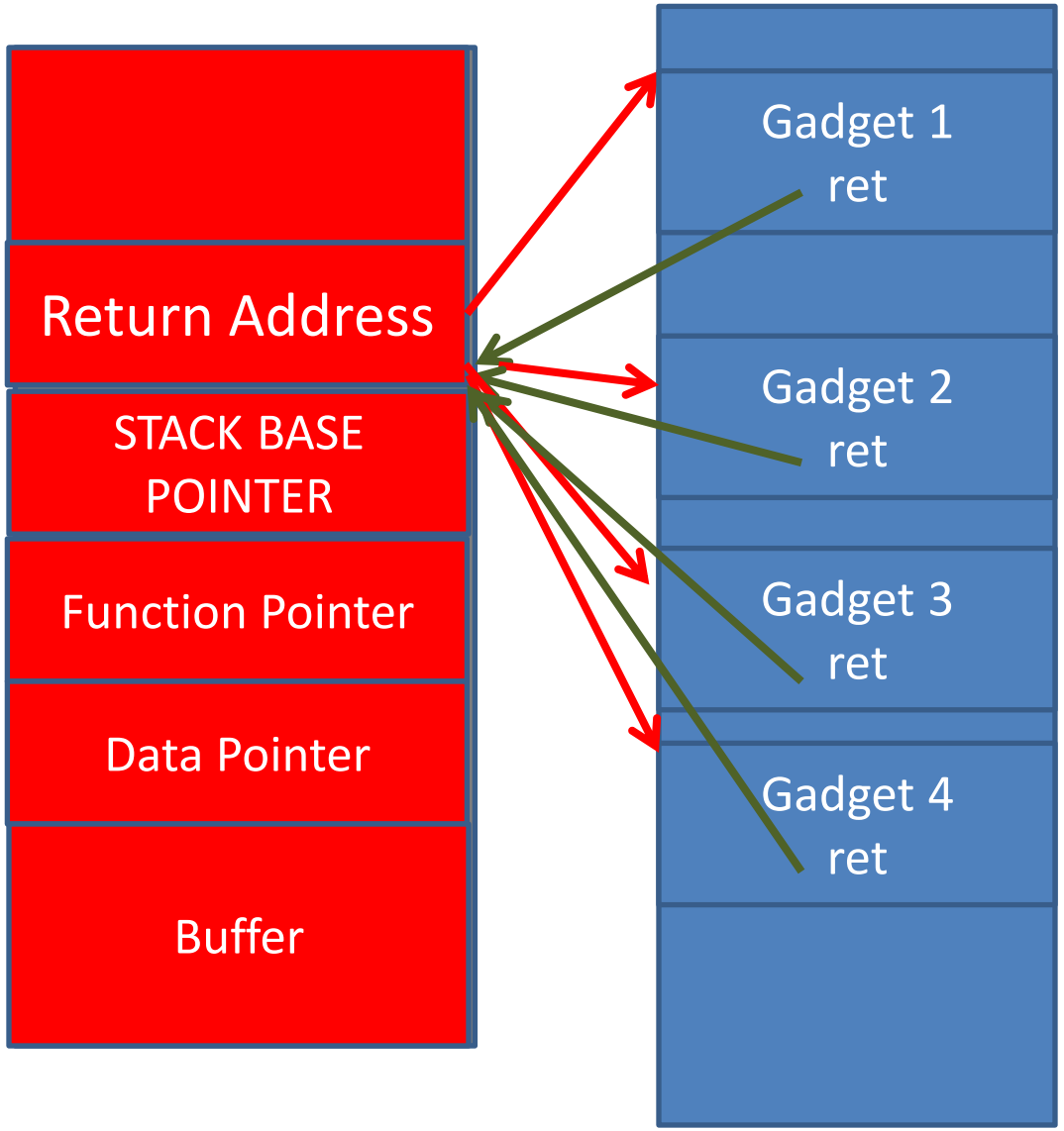
- Block Code Injection Attacks



When exploitation hits "NX"

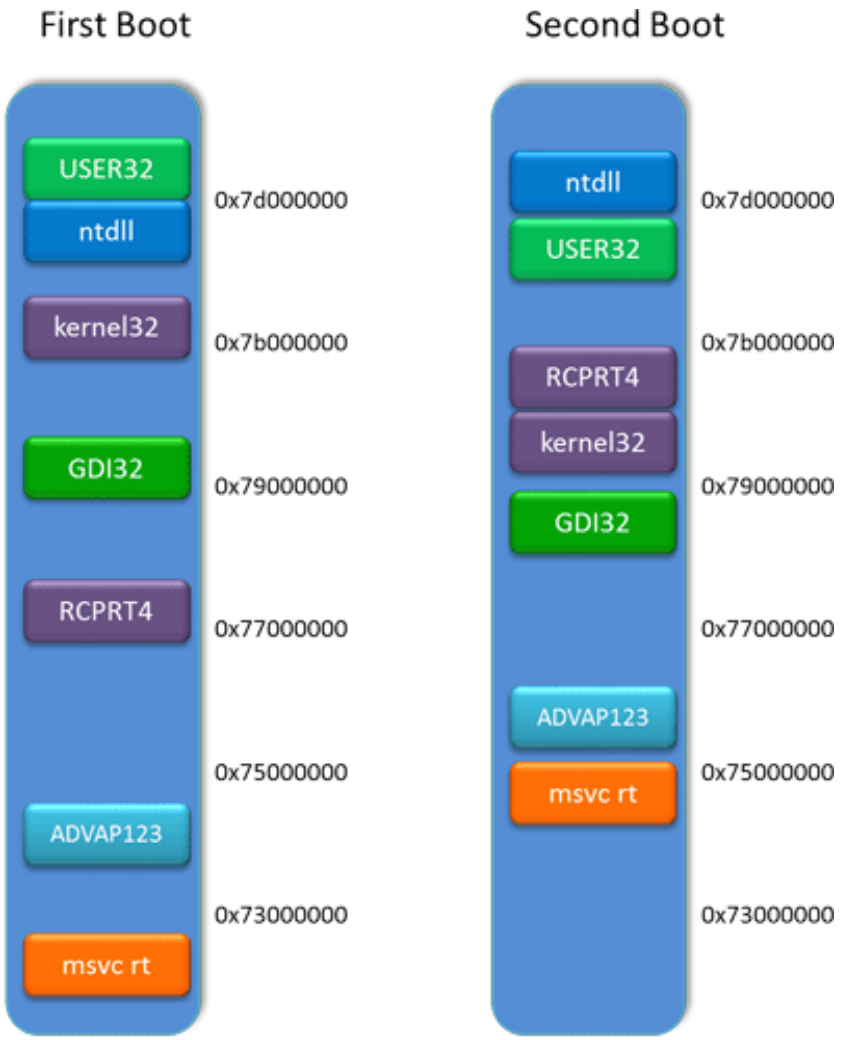
- **Return-Oriented Programming: ROP**
  - A generalized form of "return-to-libc" attack
  - Execute existing instruction sequences, in a chained "gadgets" where each gadget ends with ret

R. Roemer, E. Buchanan, H. Shacham, and S. Savage. ***Return-Oriented Programming: Systems, Languages, and Applications****. ACM Trans. Info. & Sys. Security 15(1):2, Mar. 2012*



When exploitation hits ASLR

- *Address Space Layout Randomization: ASLR*
  - *Change program internal state*
  - *Break attacker's assumption of where the code & data is*
- *Fine-grained ASLR:*
  - *Also change the relative distance between objects*



How to defeat ASLR?

- *Defeat ASLR*
  - *ASLR is often for base address randomization – no change to relative distance between two objects*

Snow, Kevin Z., et al. "**Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization**." *IEEE Symposium on Security and Privacy*. 2013.

More info on “memory error arms race”

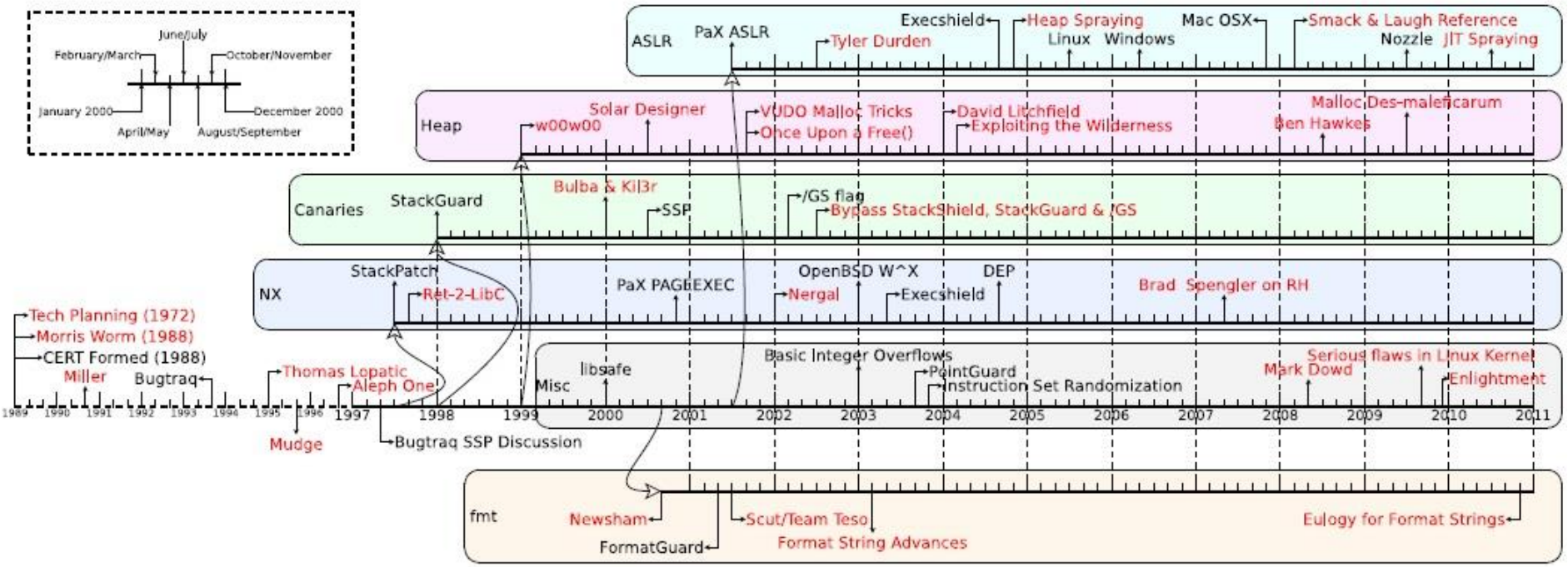


Fig. 1: General timeline

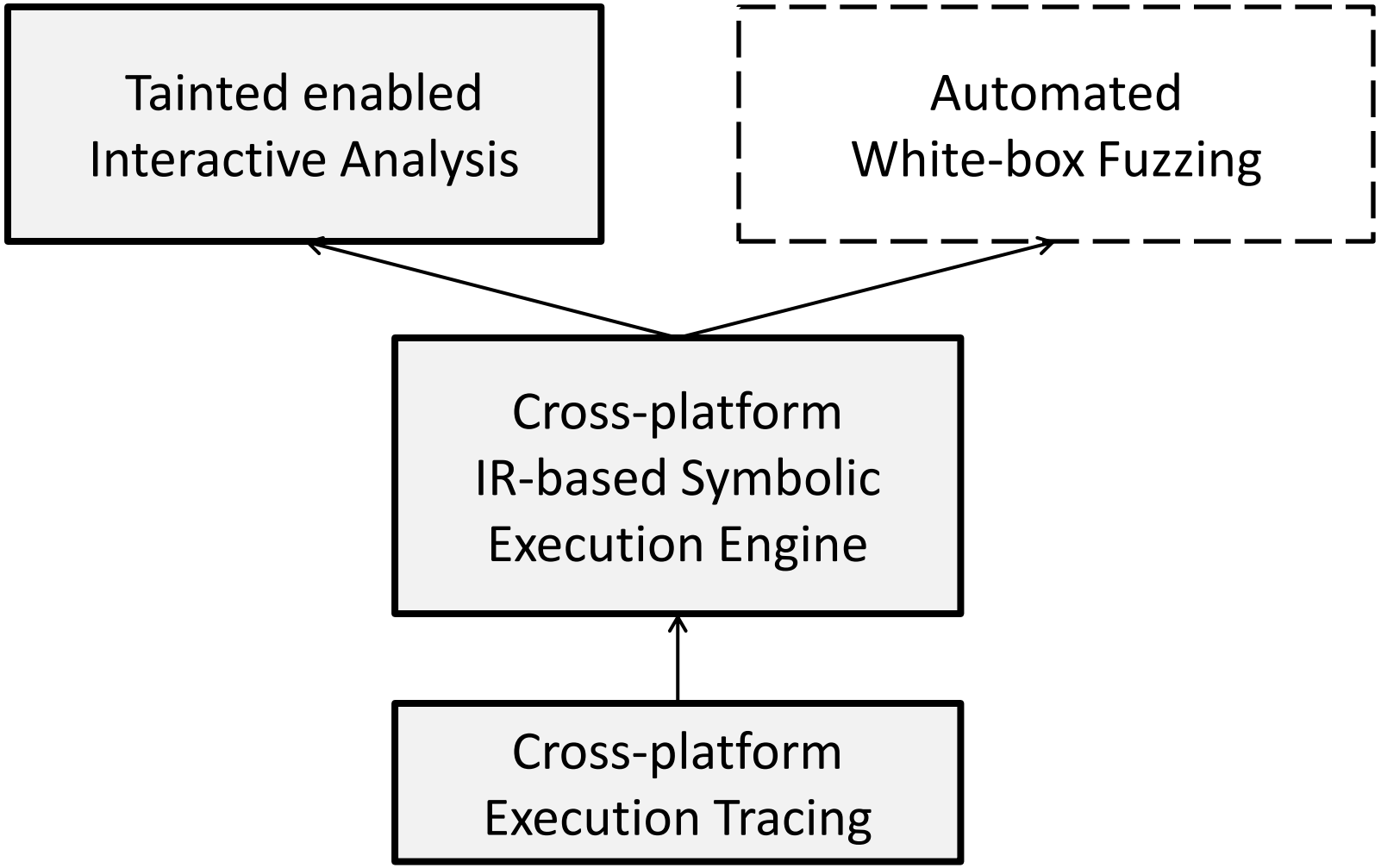
Bos, Herbert. "**Memory Errors: The Past, the Present, and the Future,”** in RAID 2012

Outline

- First Half
  - Background
    - Dynamic analysis and symbolic analysis
    - Their success in vulnerability discovery
    - Challenges in exploitation research
  - **Our analysis platform**
    - Cross-platform trace generation

- Second Half
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Our goal is to support both automated and interactive analysis



Changing Computing Landscape

Diversified Platforms

Mobile Devices

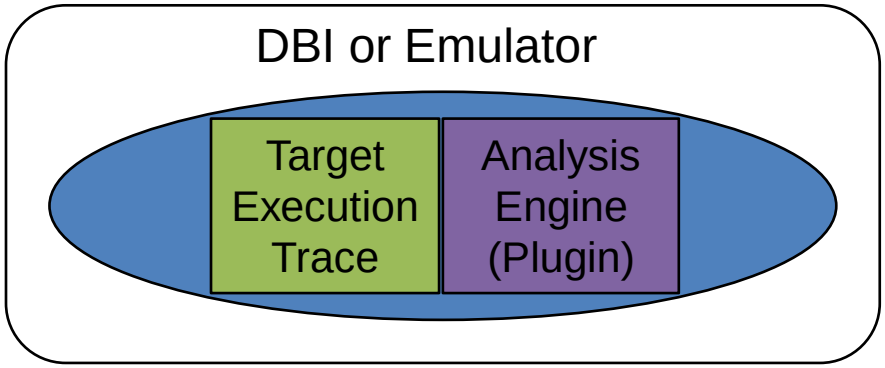
Embedded Devices

Control Systems

Cloud Computing Infrastructures

Cross-platform Analysis Problems:

- Trace analysis closely tied to trace generation and relied on instrumentation specific implementation, such as Dynamic Binary Instrumentation (DBI) or system emulator.



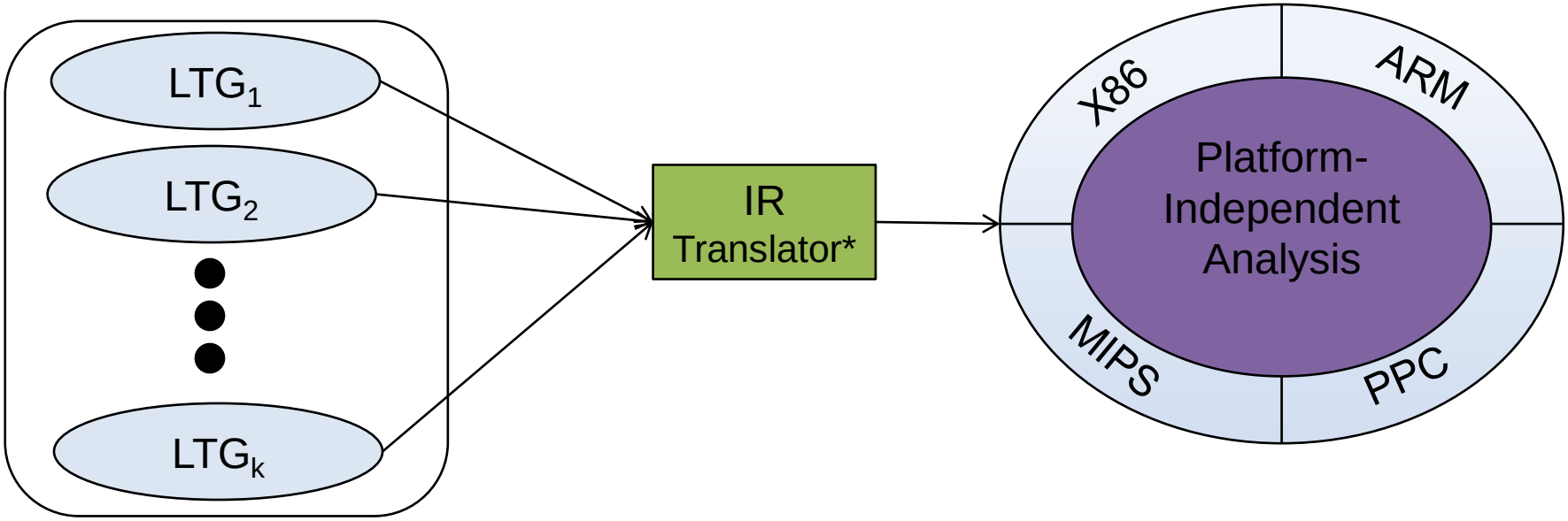
Comparisons of Popular DBI Tools:

Tool	Tool Runs On	Target ISA
PIN	Windows/Linux	X86/64 Windows/Linux
Dynamorio	Windows/Linux	X86/64 Windows/Linux
Valgrind	Linux/Android/Mac	X86/64, ARM, PPC32/64, MIPS

Our Approach:

Separate Trace Generation from Trace Analysis with Intermediate Representation

- **Platform-dependent** Light-weight Trace Generation (LTG)
- Intermediate Representation(**IR**) Translation
- **Platform-independent** Analysis Engine
- Analysis Algorithm and Implementation **Reuse**



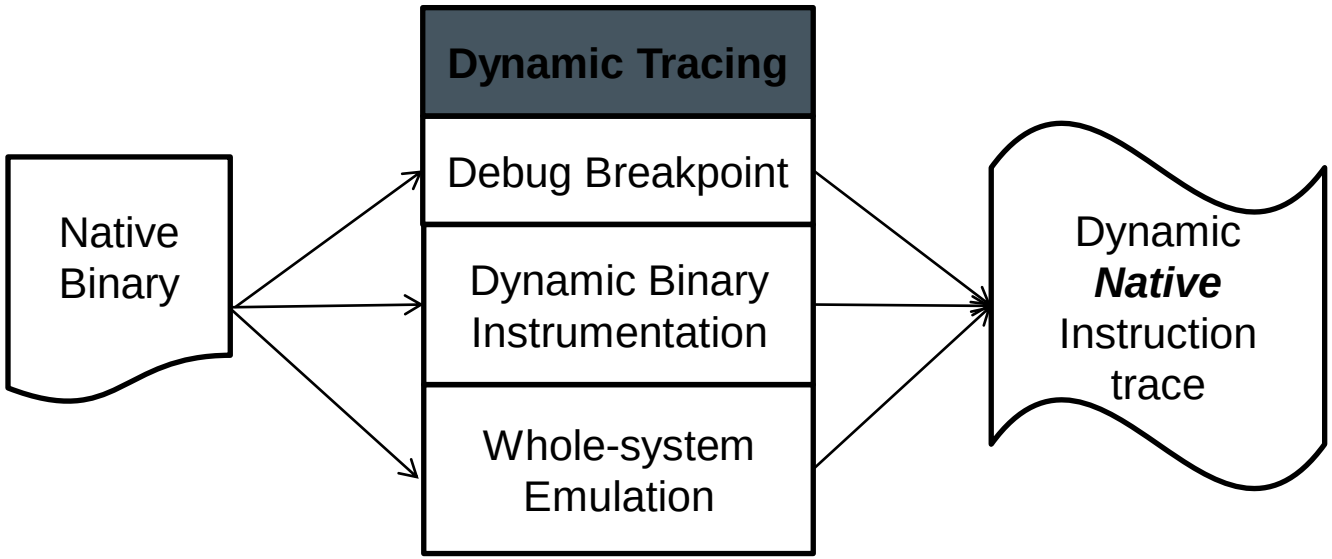
Multiple-platform Trace Generation

Trace Generation

- Dynamic Binary Instrumentation
  - Process based
  - Just-in-time binary translation
  - PIN, DynamoRIO, Valgrind
- Whole System Emulation
  - System based
  - Just-in-time binary translation
  - QEMU
- Debug Breakpoint
  - Both process(user mode) and system(kernel)
  - Universal and accessible on real devices

Light-weight Trace Generation:

- Target-dependent
- Capture basic program state using platform-specific instrumentation framework
- Support interactive tracing

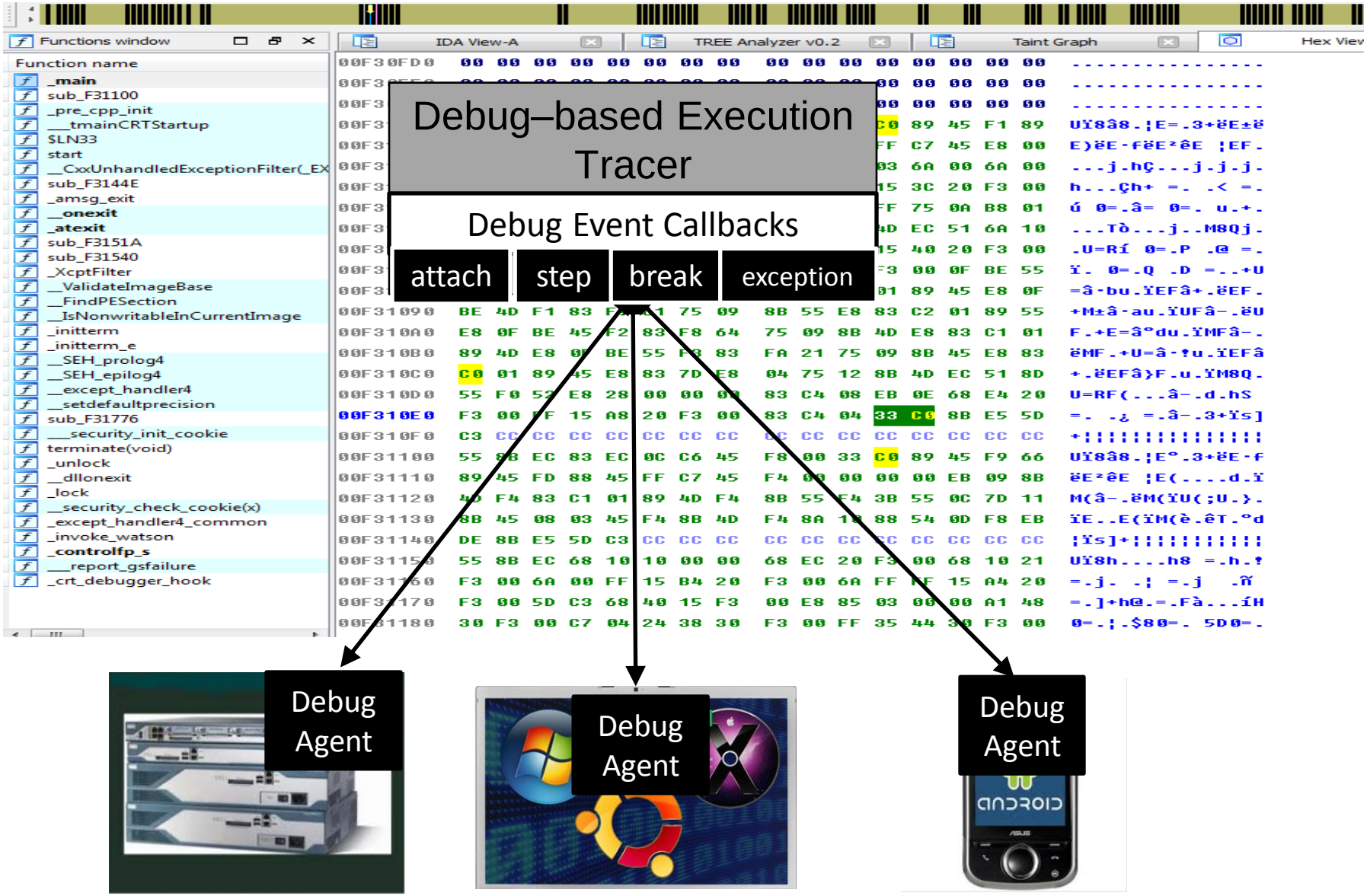


Our Debug Tracer Runs as IDA-Pro Plug-in

- **Why IDA Pro?** It is the most popular *reverse engineering tool* with cross-platform debugging support

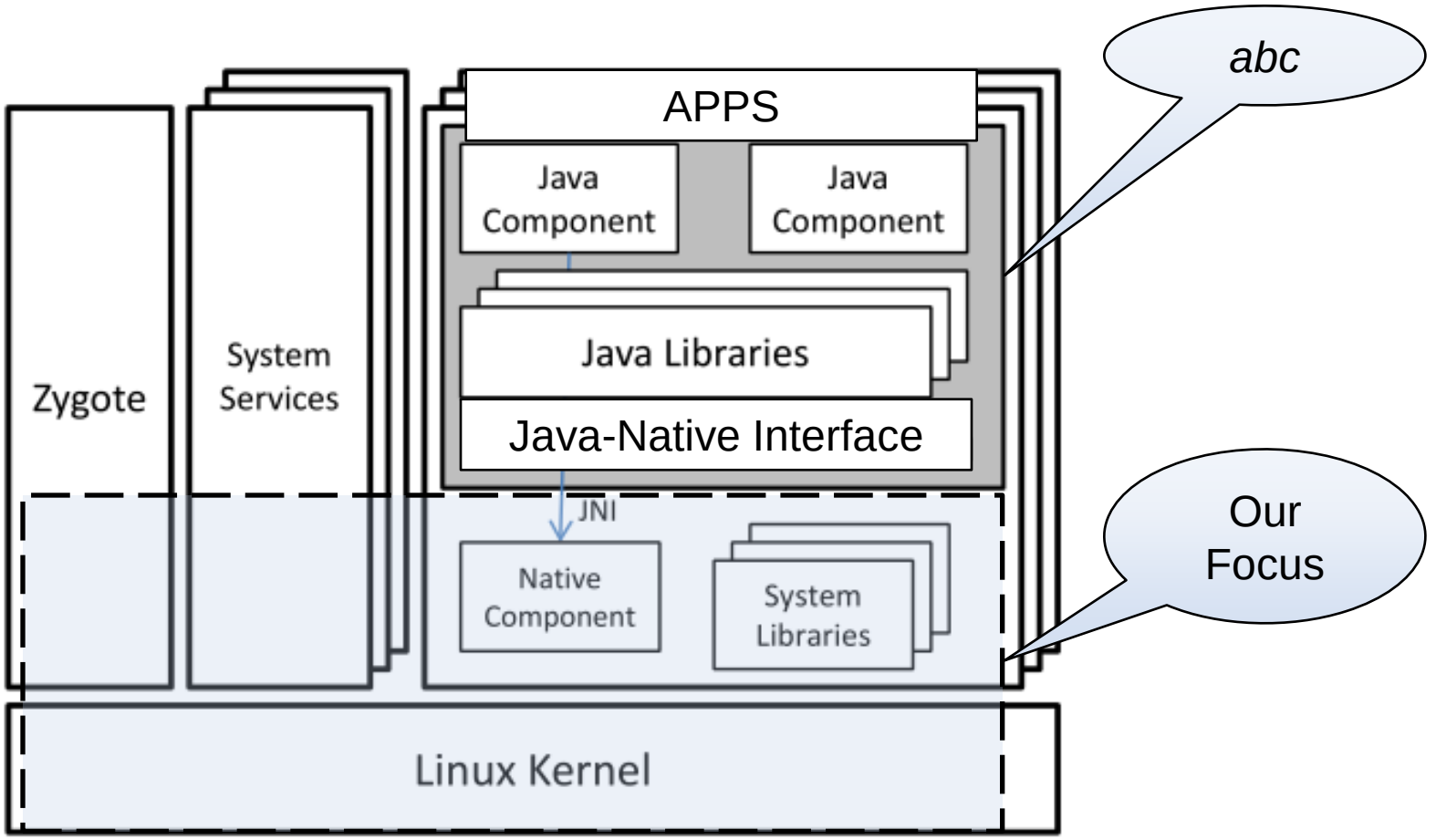
	IDA runs on: Windows	IDA runs on: Linux	IDA runs on: Mac OS X	Additional Notes
Target Platform: Windows 32/64-bit	Local/Remote	Remote	Remote	On 64-bit Windows platforms, remote only.
Target Platform: Linux 32/64-bit	Remote	Local/Remote	Remote	On 64-bit Linux platforms, remote only.
Target Platform: OS X 32/64-bit	Remote	Remote	Local/Remote	On 64-bit Mac OS X platforms, remote only.
Target Platform: iPhone	Remote	/	/	Discontinued in IDA v5.6
Target Platform: Bochs	Bochs Emulator	Bochs Emulator	Bochs Emulator	
Target Platform: GDB Server	GDB Server	GDB Server	GDB Server	Debugger based on GDB Server. Particularly useful for windows or linux kernel debugging via VMWare, or even QEMU-based debugging. Currently supported processors: x86, ARM, PowerPC, MIPS.
Target Platform: WinDBG 32/64-bit	Remote	/	/	Both user-mode and kernel-mode debugging are available. 64-bit debugging is supported too. See help page

Our Debug Execution Tracer



Multiple-platform Tracing on Android Demo

Android Architecture

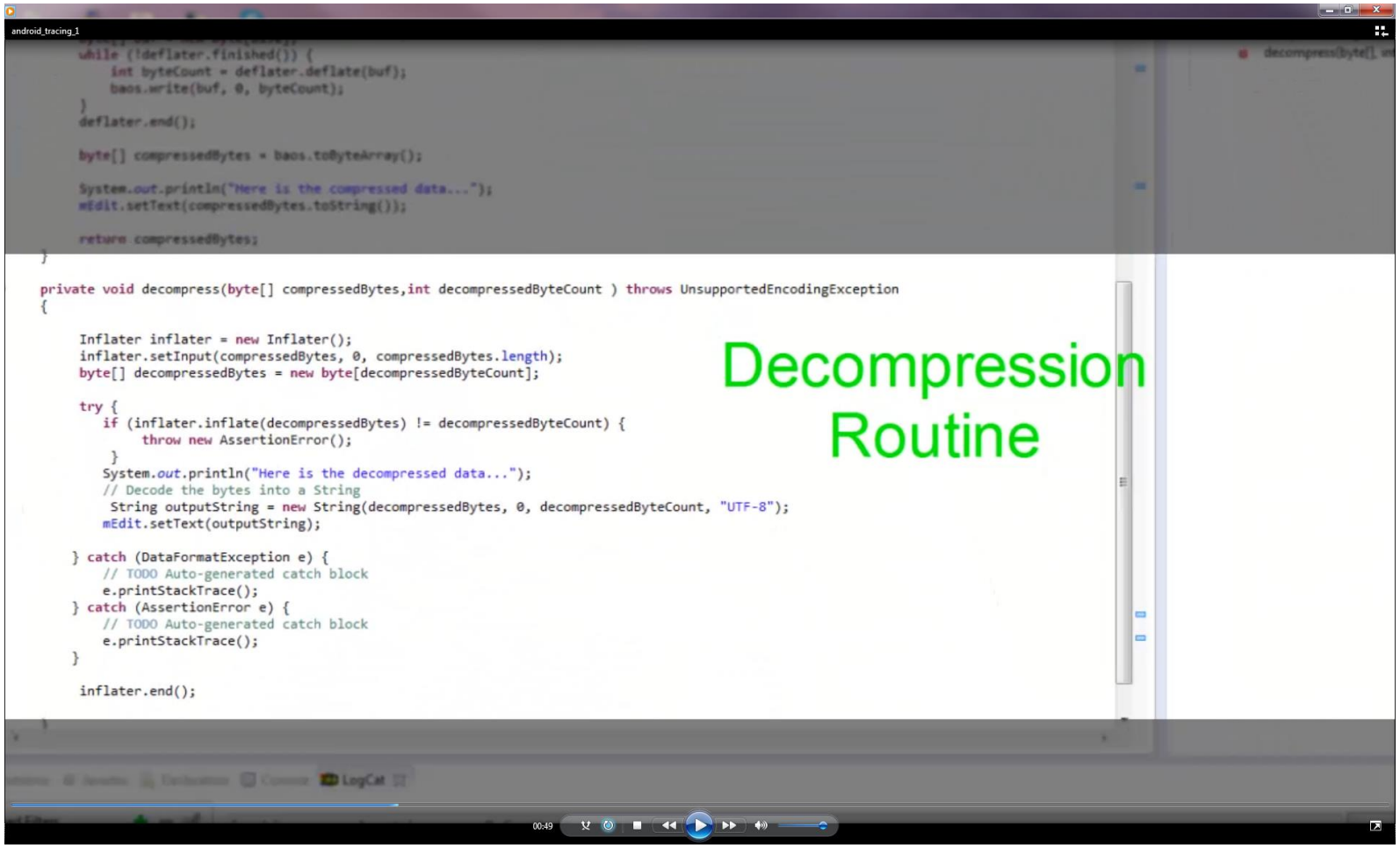


Android Tracing Setup

Trace Generation on Android-based Device

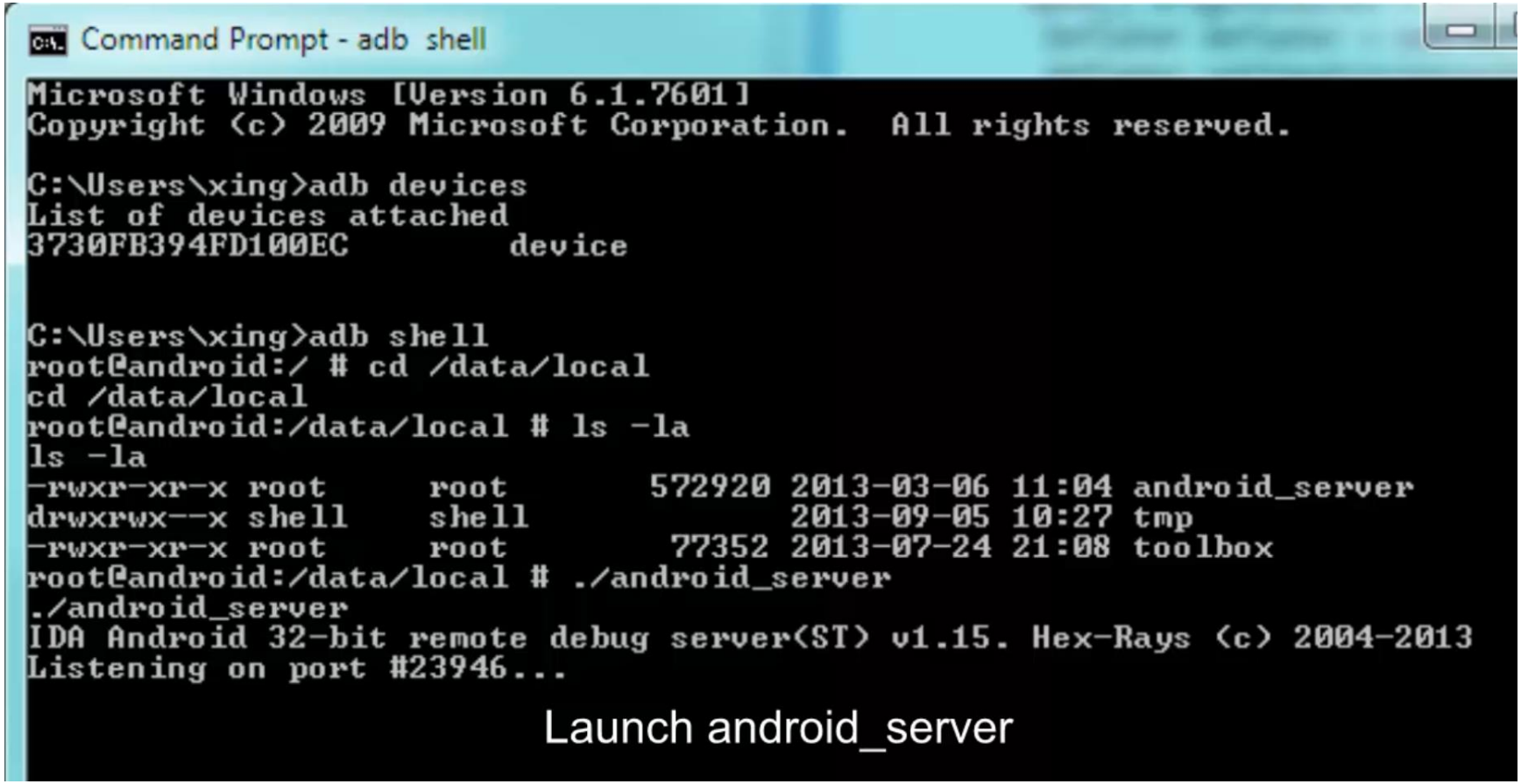
Trace Zip code in zlib.so

Key Step: Create an APP to trigger the compression/decompression activity



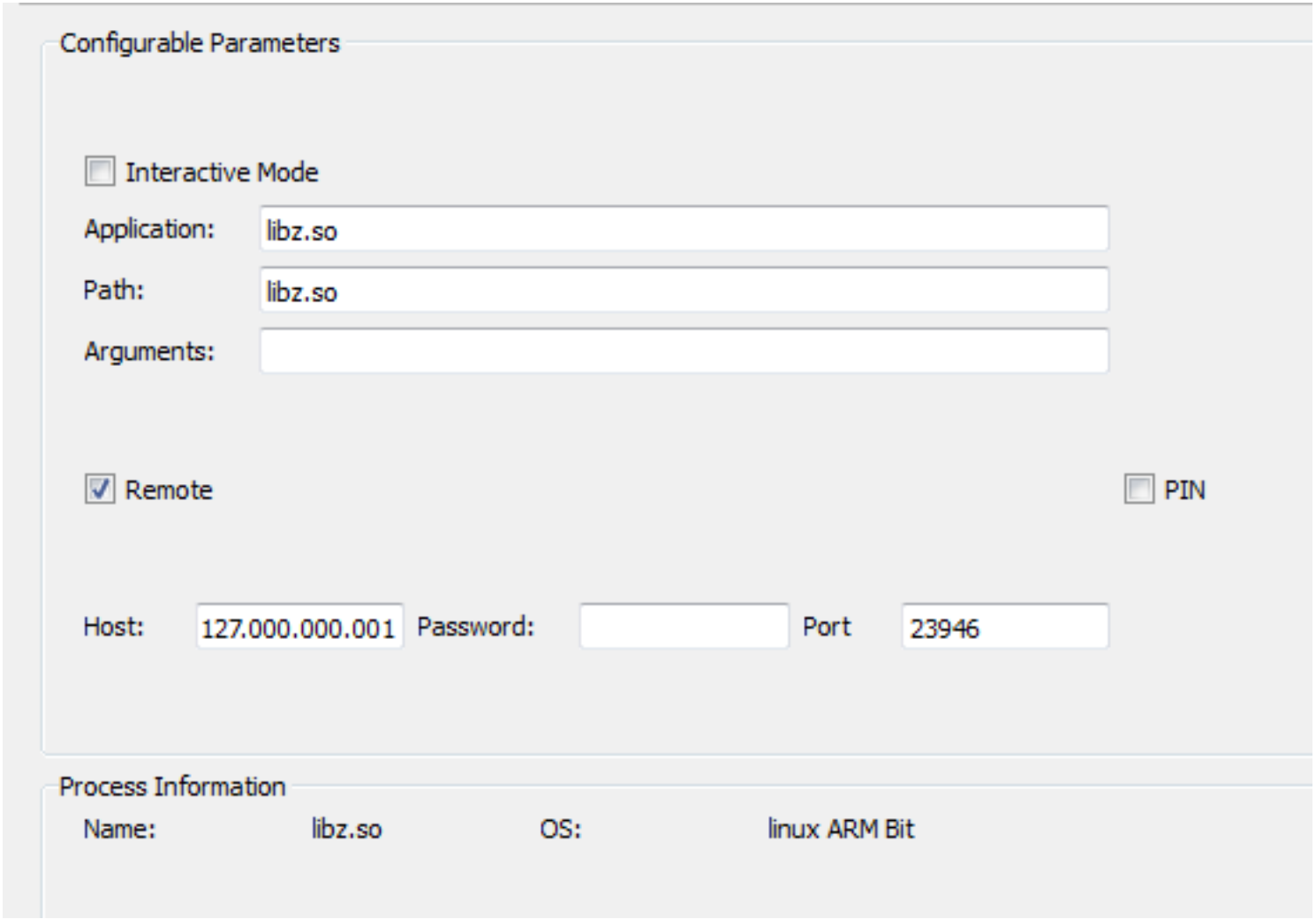
Trace Zip code in zlib.so (cont)

Key Step: Launch Debug Agent on the Device



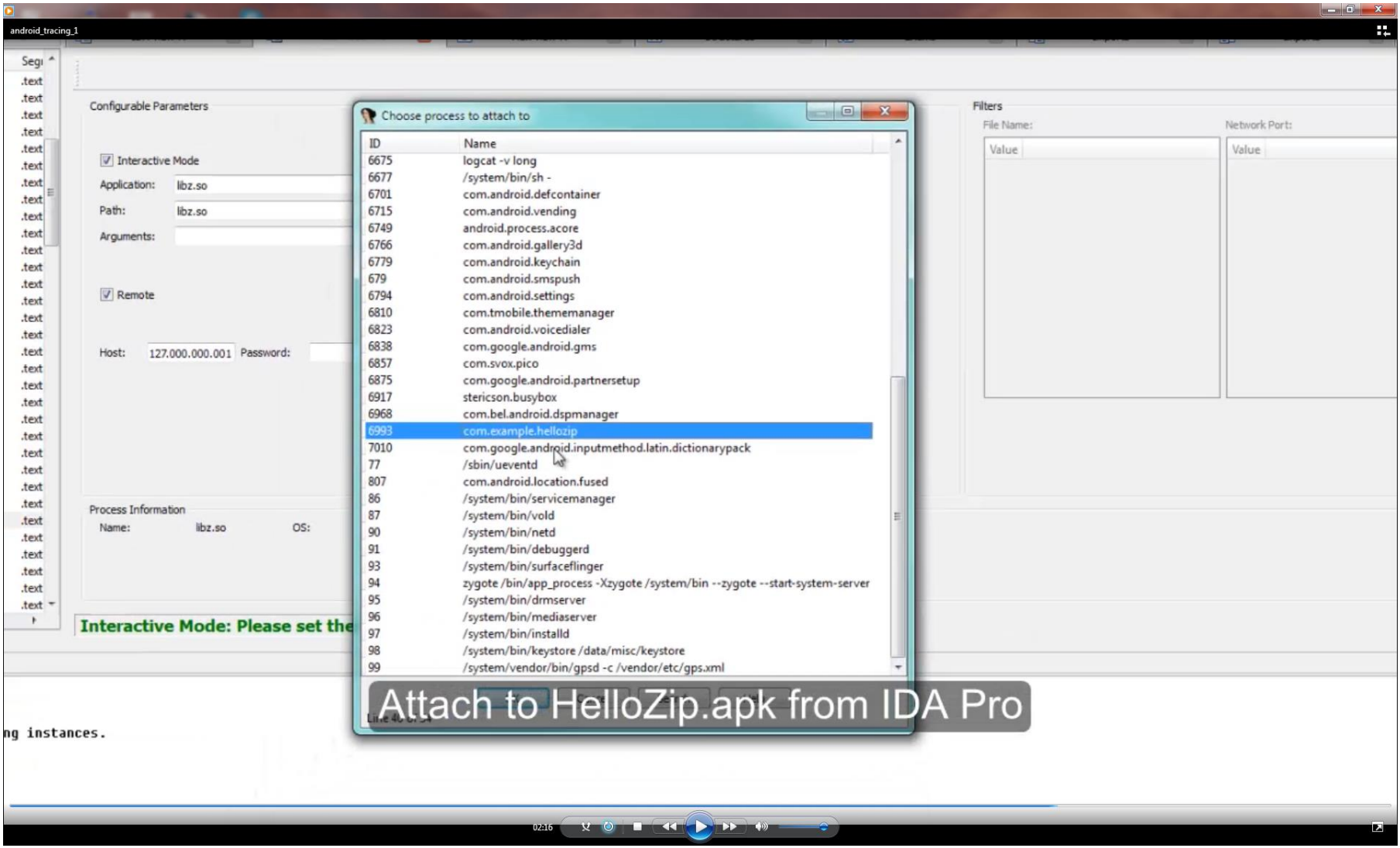
Trace Zip code in zlib.so (cont)

Key Step: Select the tracing mode and parameters



Trace Zip code in zlib.so (cont)

Key Step: Attach tracer to the right process



Trace Zip code in zlib.so (cont)

Key step: Collect program states from callback

```
1  E 0x40037208 4 f04f2de9 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
2  E 0x4003720c 4 005050e2 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
3  E 0x40037210 4 e88b9fe5 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
4  E 0x40037214 4 1cd04de2 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
5  E 0x40037218 4 0160a0e1 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
6  E 0x4003721c 4 08808fe0 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
7  E 0x40037220 4 e801000a R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
8  E 0x40037224 4 1c4095e5 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x402
9  E 0x40037228 4 013074e2 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x4014c228, R4=0x4d5
10 E 0x4003722c 4 0030a033 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0xb2a33ff9, R4=0x4d5
11 E 0x40037230 4 050051e3 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x0, R4=0x4d5cc008,
12 E 0x40037234 4 013083c3 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x0, R4=0x4d5cc008,
13 E 0x40037238 4 000053e3 R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x0, R4=0x4d5cc008,
14 E 0x4003723c 4 e101001a R0=0x4d5d8048, R1=0x4, R2=0x2000, R3=0x0, R4=0x4d5cc008,
```

Demo

Android Trace Generation Demo

Halftime

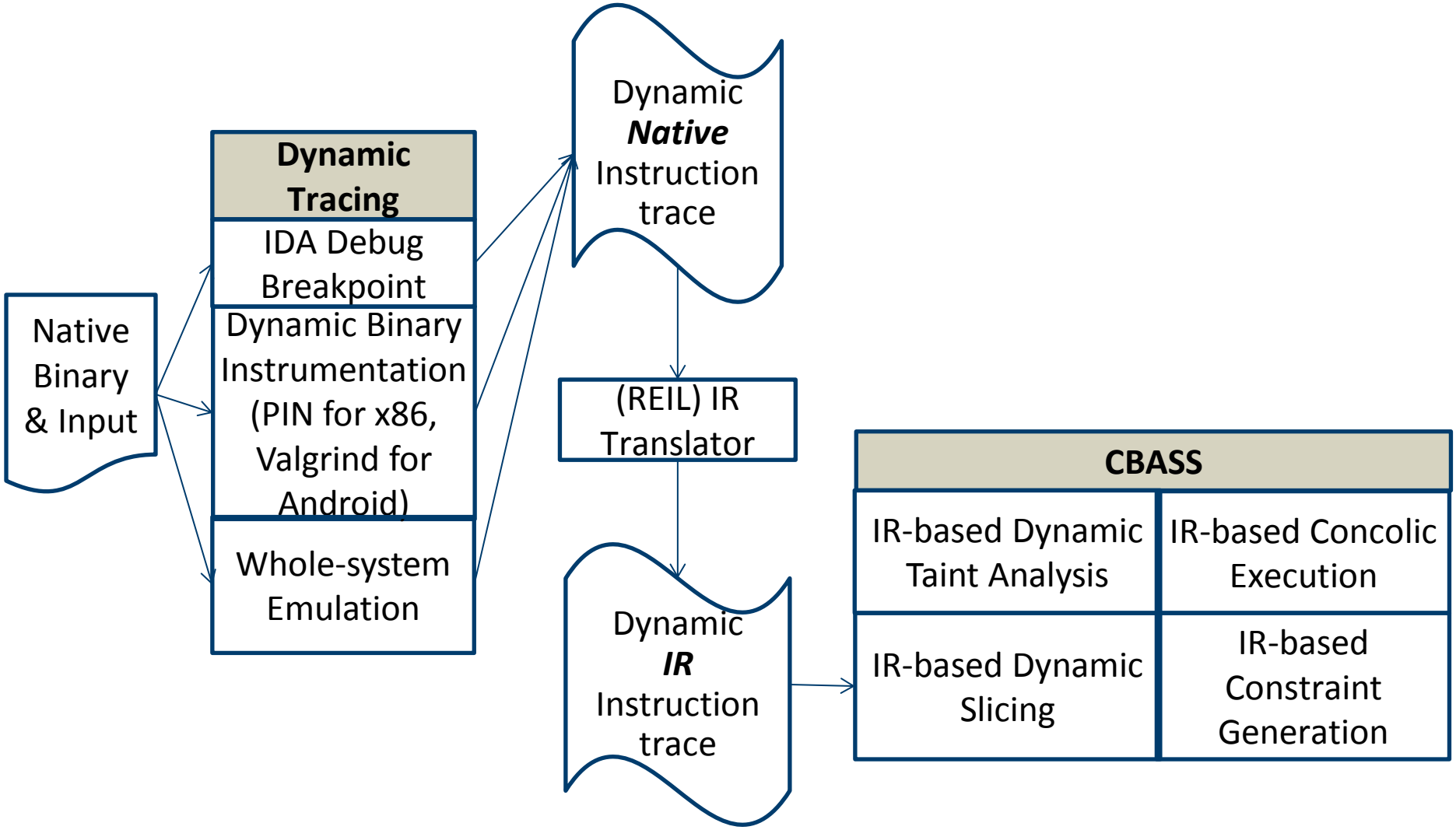
Outline

- First Half
  - Background
    - Dynamic analysis and symbolic analysis
    - Their success in vulnerability discovery
    - Challenges in exploitation research
  - Our analysis platform
    - Cross-platform trace generation

- Second Half
  - **Supporting automated analysis**
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Architecture (ISA) Independent Concrete-Symbolic (Concolic) Execution

Cross-platform Tracing and IR-based Symbolic Execution Engine



Intermediate Representation

- Prerequisite:
  - Binary code stripped of symbols

- Requirements for IR:
  - Translators are already available for popular ISAs:
    - X86, ARM, PowerPC and MIPS
  - Representation is simple and low level
    - Easy mapping from native to IR instructions
    - Can leverage program state from native trace
  - The semantics of IR instructions amenable to symbolic analysis

- REIL (Reverse Engineering Intermediate Language)
  - http://www.zynamics.com/binnavi/manual/html/reil_language.htm

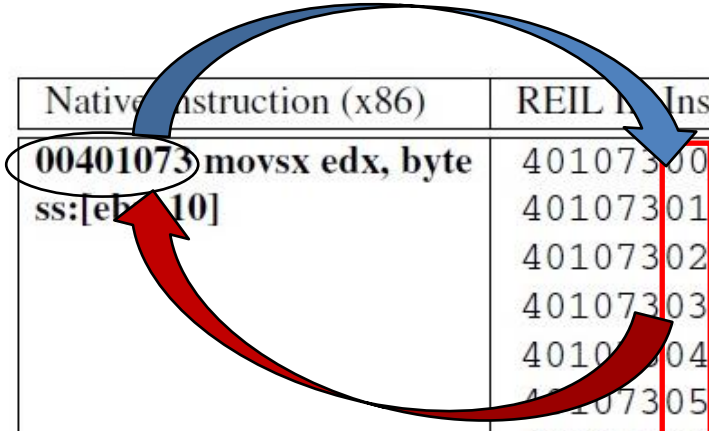
Reverse Engineering Intermediate Language (REIL) Instruction Set

Category	REIL Instruction	Semantics
Arithmetic	ADD s1, s2, d	d = s1 + s2
	SUB s1, s2, d	d = s1 s2
	MUL s1, s2, d	d = s1 ∗ s2
	DIV s1, s2, d	d = s1/s2
	MOD s1, s2, d	d = s1 mod s2
	BSH s1, s2, d	if s2>0 d = s1∗2^{s2}
		else d = s1 /2^{-s2}
Bitwise	AND s1, s2, d	d = s1& s2
	OR s1, s2, d	d = s1 | s2
	XOR s1, s2, d	d = s1 xor s2
Logical	BISZ s1, ∉, d	if s1 = 0, d = 1 else d = 0
	JCC s1, ∉, d	iff s1≠0, set eip = d
Transfer	LDM s1, ∉, d	d = mem[s1]
	STM s1, ∉, d	mem[d] =s1
	STR s1, ∉, d	d = s1
Other	NOP, ∉, ∉, ∉	No op
	UNDEF ∉, ∉,d	Undefined instruction
	UNKN ∉, ∉, ∉	Unknown instruction

Benefits of REIL: The Easy Mapping from Native Instructions to REIL IR

Original Native Address **->** Shift 1 Byte to the Left + Offset **->** REIL IR Address

REIL IR Address **->** Shift 1 Byte to the Right **->** Original Native Address



Native Instruction (x86)	REIL IR Instruction
00401073 movsx edx, byte ss:[eb 10]	40107300: add [DWORD FFFFFFF0, DWORD ebp, QWORD t0] 40107301: and [QWORD t0, DWORD FFFFFFFF, DWORD t1] 40107302: ldm [DWORD t1, EMPTY , BYTE t2] 40107303: xor [BYTE t2, BYTE 0x80, BYTE t3] 4010 304: sub [BYTE t3, BYTE 0x80, DWORD t4] 40107305: and [DWORD t4, BYTE FFFFFFFF, BYTE t5] 40107306: str [DWORD t5, EMPTY , DWORD edx]

Benefits of REIL: (cont)
REIL makes native instruction side effects explicit

X86 *Sub* instruction semantics

Operation:

DEST <- DEST – SRC;

Description (from Intel Manual)

This instruction subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

Native Instruction: sub esi, ss:[esp+12]
Translated to the following REIL IRs:

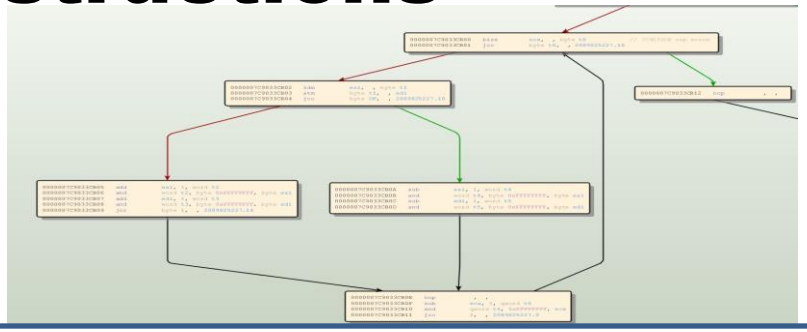
```
add [DWORD 12, DWORD esp, QWORD t0]
and [QWORD t0, DWORD 4294967295, DWORD t1]
ldm [DWORD t1, EMPTY , DWORD t2]
and [DWORD esi, DWORD 2147483648, DWORD t3]
and [DWORD t2, DWORD 2147483648, DWORD t4]
sub [DWORD esi, DWORD t2, QWORD t5]
and [QWORD t5, QWORD 2147483648, DWORD t6]
bsh [DWORD t6, DWORD -31, BYTE SF]
xor [DWORD t3, DWORD t4, DWORD t7]
xor [DWORD t3, DWORD t6, DWORD t8]
and [DWORD t7, DWORD t8, DWORD t9]
bsh [DWORD t9, DWORD -31, BYTE OF]
and [QWORD t5, QWORD 4294967296, QWORD t10]
bsh [QWORD t10, QWORD -32, BYTE CF]
and [QWORD t5, QWORD 4294967295, DWORD t11]
bisz [DWORD t11, EMPTY , BYTE ZF]
str [DWORD t11, EMPTY , DWORD esi]
```

Benefits of REIL: (cont)
REIL IR can enable static analysis on complex native instructions

MOVS/MOVSB/MOVSW/MOVSD—Move Data from String to String
REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix:

Operation

```
IF AddressSize = 16
   THEN
      use CX for CountReg;
   ELSE (* AddressSize = 32 *)
   use ECX for CountReg;
FI;
WHILE CountReg <> 0
   DO
        service pending interrupts (if any);
        execute associated string instruction;
        CountReg <- CountReg – 1;
        IF CountReg = 0
           THEN exit WHILE loop
        FI;
        IF (repeat prefix is REPZ or REPE) AND (ZF=0)
        OR (repeat prefix is REPNZ or REPNE) AND (ZF=1)
            THEN exit WHILE loop
        FI;
OD;
```



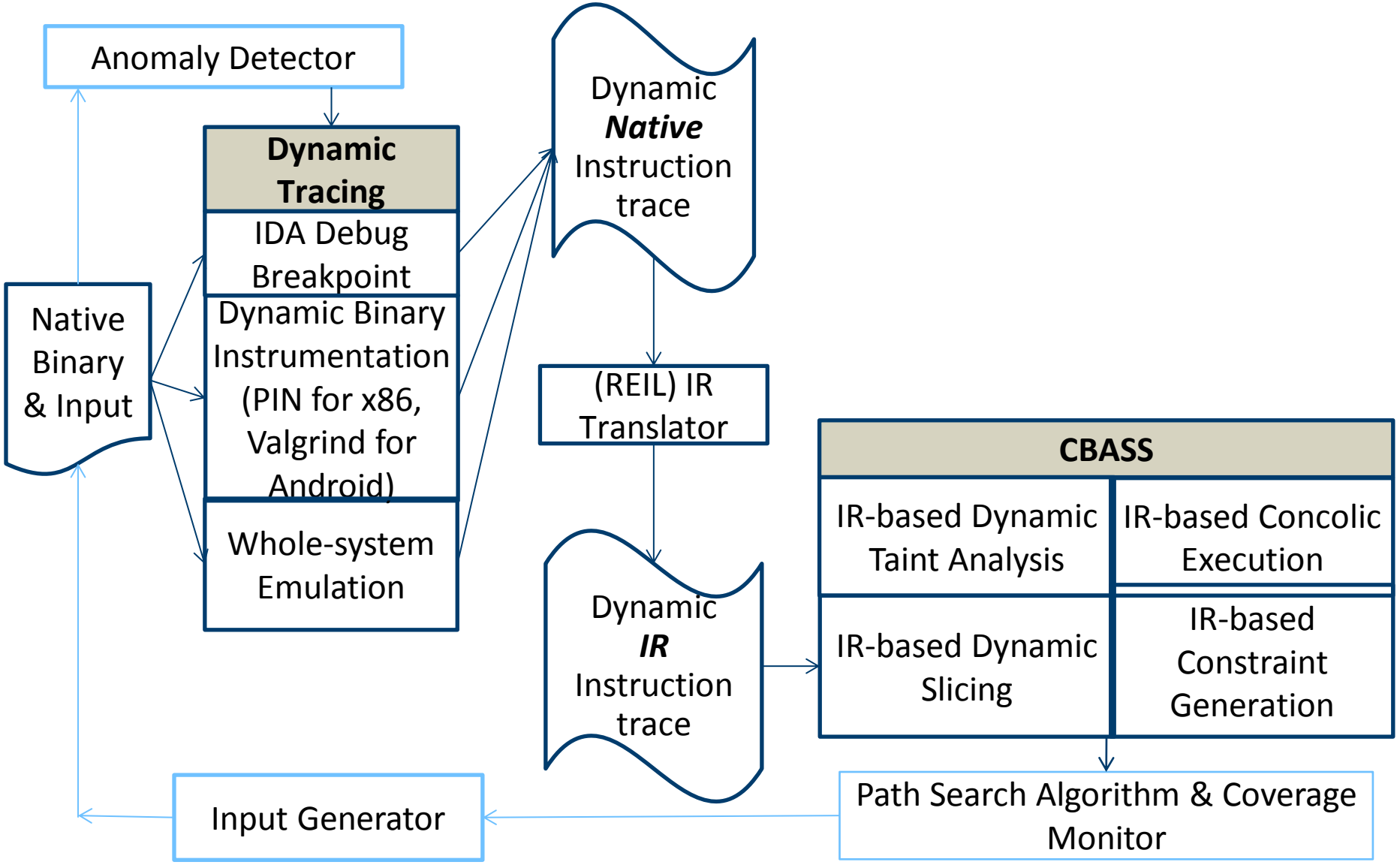
Native Instruction: 7c9033cb rep movsb IRs:

7C9033CB00: bisz [DWORD ecx, EMPTY , BYTE t0]
7C9033CB01: jcc [BYTE t0, EMPTY , ADDRESS 7C9033CB12]
7C9033CB02: ldm [DWORD esi, EMPTY , BYTE t1]
7C9033CB03: stm [BYTE t1, EMPTY , DWORD edi]
7C9033CB04: jcc [BYTE DF, EMPTY , ADDRESS 7C9033CB10]
7C9033CB05: add [DWORD esi, DWORD 1, WORD t2]
7C9033CB06: and [WORD t2, BYTE 4294967295, BYTE esi]
7C9033CB07: add [DWORD edi, DWORD 1, WORD t3]
7C9033CB08: and [WORD t3, BYTE 4294967295, BYTE edi]
7C9033CB09: jcc [BYTE 1, EMPTY , ADDRESS 7C9033CB0E]
7C9033CB0A: sub [DWORD esi, DWORD 1, WORD t4]
7C9033CB0B: and [WORD t4, BYTE 4294967295, BYTE esi]
7C9033CB0C: sub [DWORD edi, DWORD 1, WORD t5]
7C9033CB0D: and [WORD t5, BYTE 4294967295, BYTE edi]
7C9033CB0E: nop [EMPTY , EMPTY , EMPTY]
7C9033CB0F: sub [DWORD ecx, DWORD 1, QWORD t6]
7C9033CB10: and [QWORD t6, DWORD 4294967295, DWORD ecx]
7C9033CB11: jcc [DWORD 1, EMPTY , ADDRESS 7C9033CB00]
7C9033CB12: nop [EMPTY , EMPTY , EMPTY]

Concolic Execution on REIL IR Using SMT Operations

Native Instructions	REIL Instructions	Symbolic Execution, with ebp = 0x12ff84 and mem[12ff74] = INPUT
00401073 movsx edx, byte ss:[ebp-10]	40107300: add [DWORD FFFFFFF0, DWORD ebp, QWORD t0]	t0 = 0x12ff84+0xfffffff0 = 10012ff74
	40107301: and [QWORD t0, DWORD FFFFFFFF, DWORD t1]	t1 = t0 and 0xffffffff =0x12ff74
	40107302: ldm [DWORD t1, EMPTY , BYTE t2]	t2 = mem[t1] =INPUT_VAR[8]
	40107303: xor [BYTE t2, BYTE 0x80, BYTE t3]	t3 = INPUT_VAR[8] xor 0x80
	40107304: sub [BYTE t3, BYTE 0x80, DWORD t4]	t4 = (INPUT_VAR[8] xor 0x80) -0x80
	40107305: and [DWORD t4, BYTE FFFFFFFF, BYTE t5]	t5 = ((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff
	40107306: str [DWORD t5, EMPTY , DWORD edx]	edx = ((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff
00401077 cmp edx, 0x62	40107700: and [DWORD edx, DWORD 0x80000000, DWORD t0]	t0 = (((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff)and 0x80000000
	40107701: and [DWORD 98, DWORD 0x80000000, DWORD t1]	t1 = 98 and 0x80000000 = 98
	40107702: sub [DWORD edx, DWORD 98, QWORD t2]	t2 = (((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff) - 98
	Ignore irrelevant temps ...	...
	4010770B: and [QWORD t2, QWORD FFFFFFFF, DWORD t8]	t8= ((((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff) 98) and 0xffffffff
	4010770C: bisz [DWORD t8, EMPTY , BYTE ZF]	ZF = ite(t8==0,1,0)
0040107a jnz loc_40108e	40107A00: bisz [BYTE ZF, EMPTY , BYTE t0]	t0 = ite(ZF==0,1,0)
	40107A01: jcc [BYTE t0, EMPTY , DWORD 0x40108e]	eip = ite(t0==1,0x40108e,0x40107c)

Integrate Automated and Interactive Analysis System around CBASS

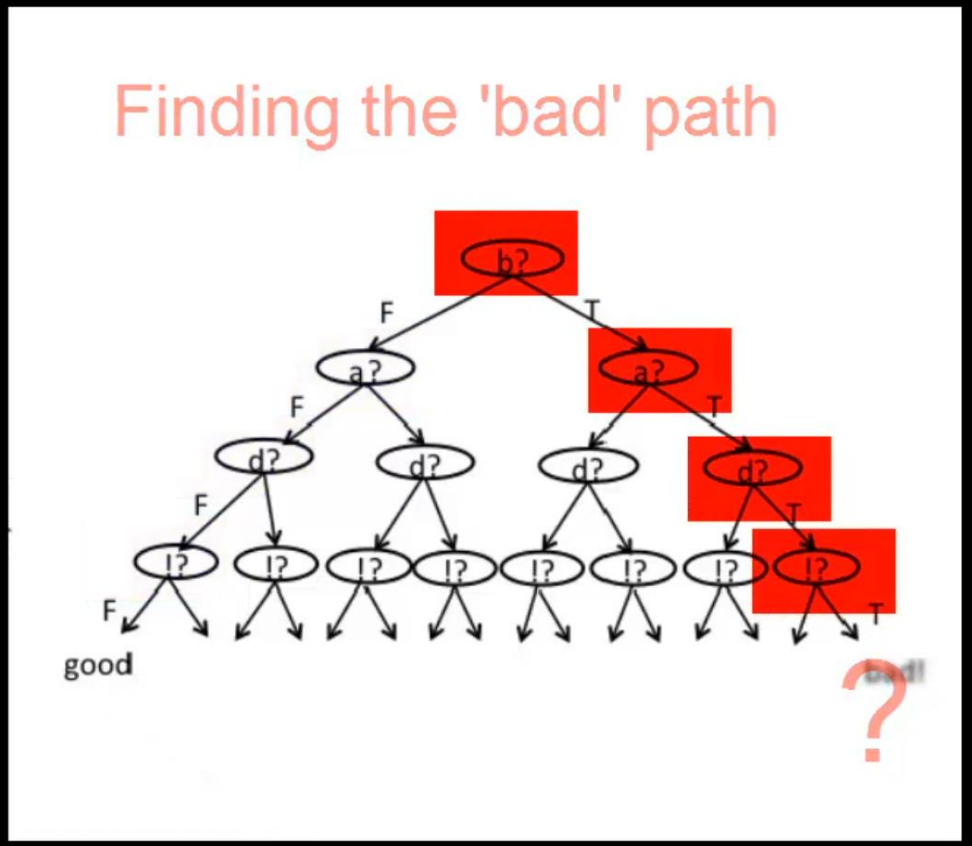


CBASS Supports Automated Symbolic Path Exploration (White-box Fuzzing)

cbass_automated_fuzz

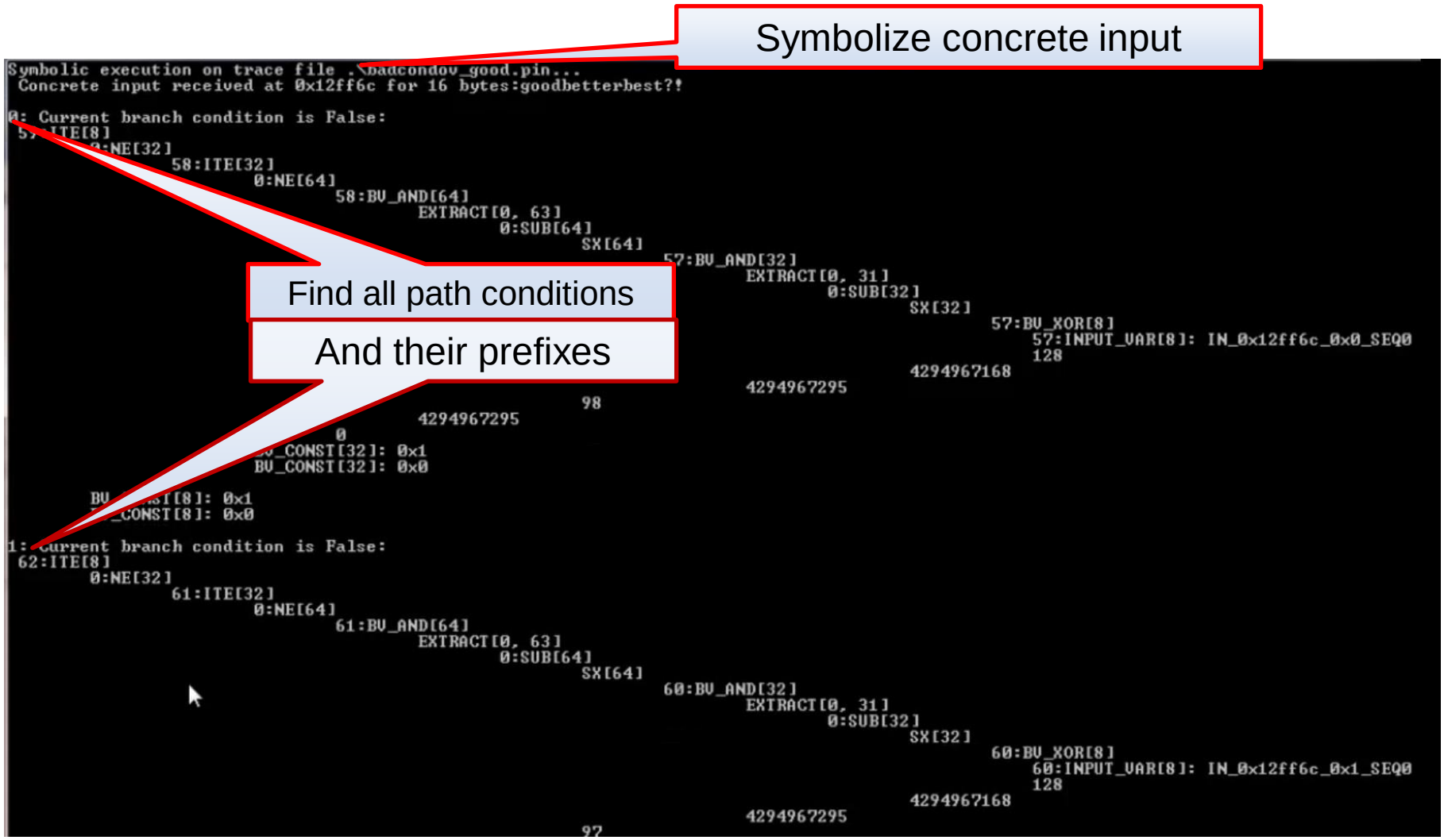
Steps:

1. Generate IDB file with IDA Pro
2. Import IDB file into BinNavi
3. Generate a 'good' trace with Pin
4. Find the 'bad' path with CBASS

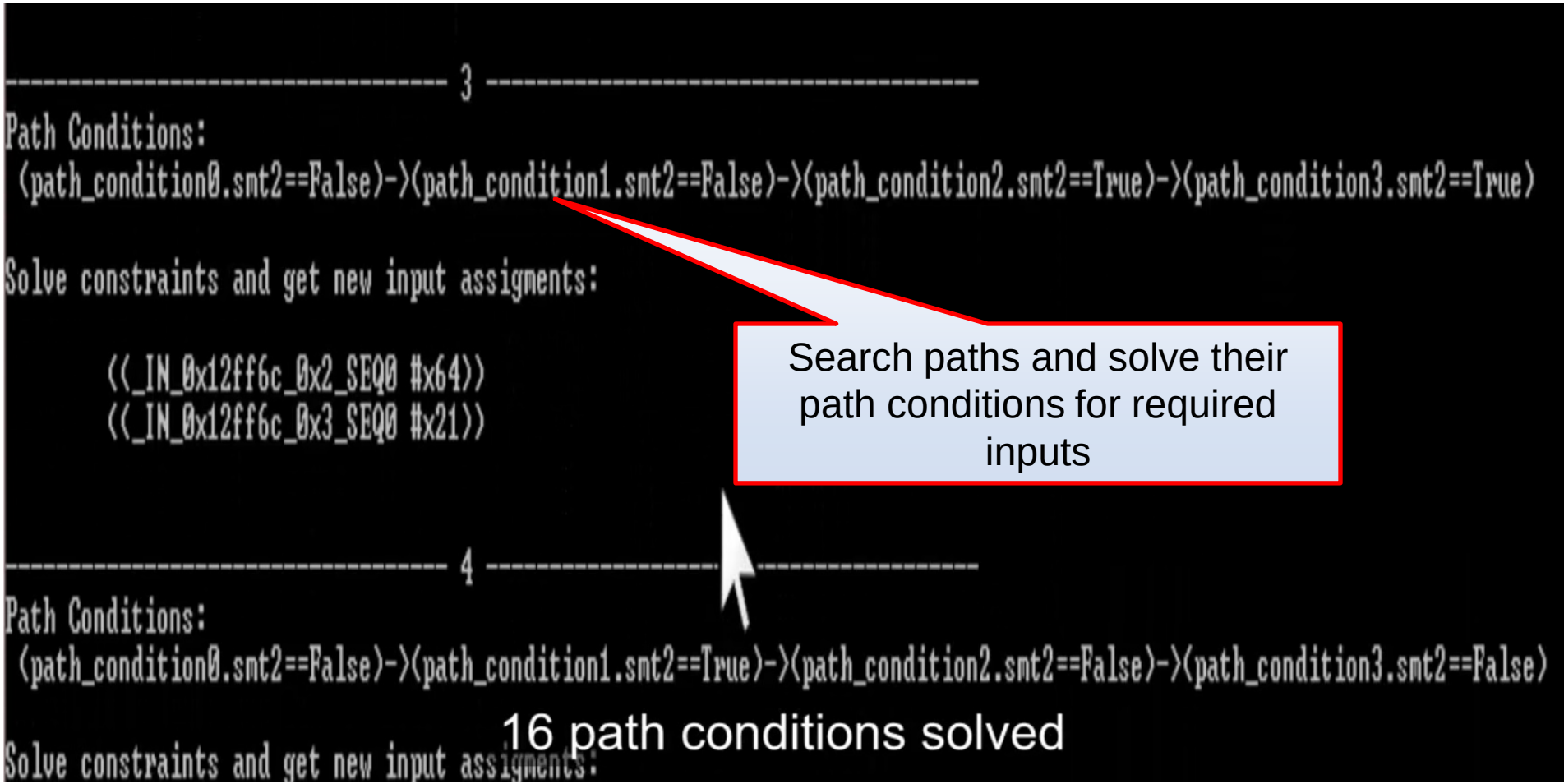


03:16

Key Steps of Automated Symbolic Path Exploration (White-box Fuzzing)



Key Steps of Automated Symbolic Path Exploration (White-box Fuzzing)



Automated White-box Fuzzing with CBASS

Demo

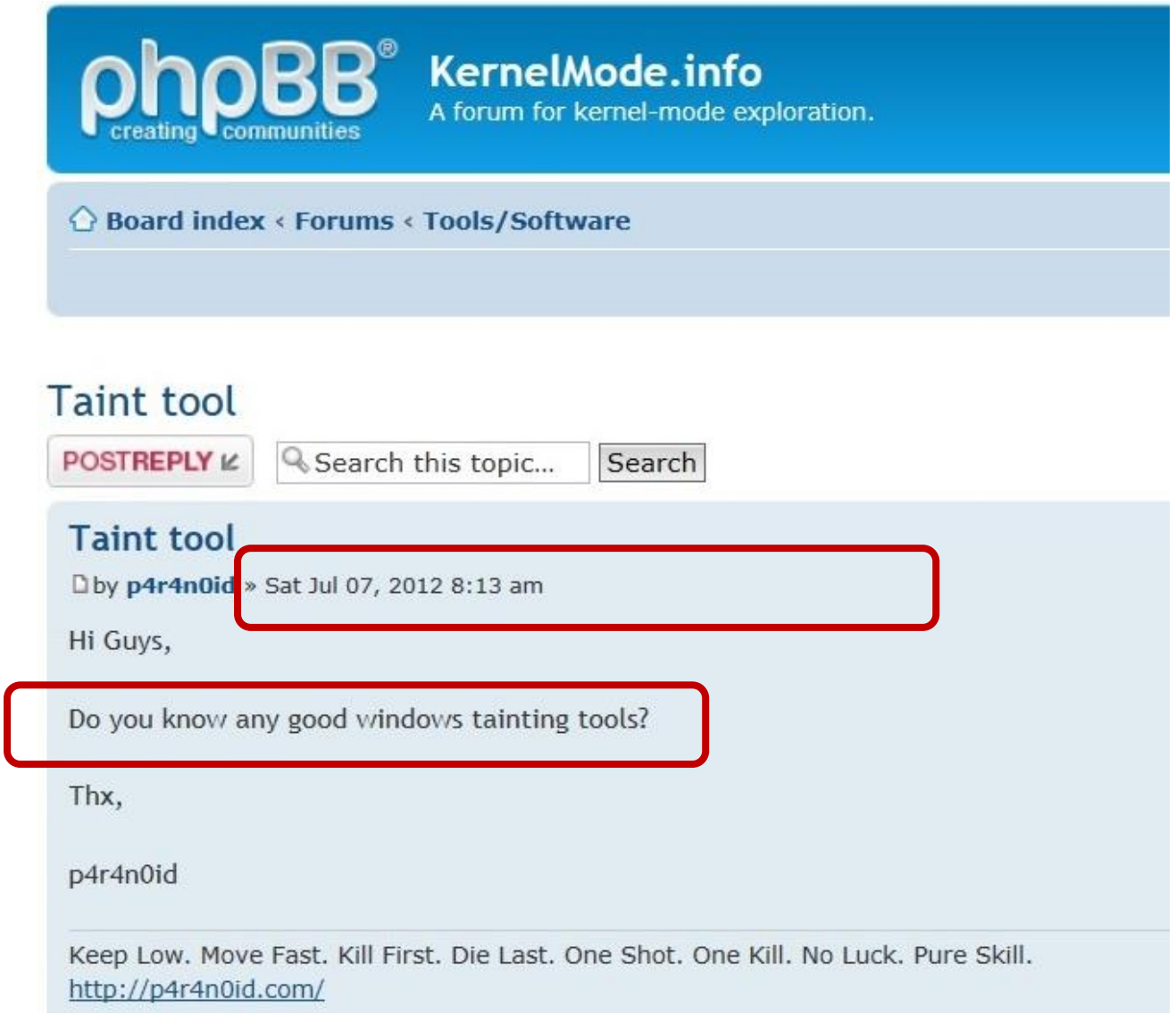
Outline

- First Half
  - Background
    - Dynamic analysis and symbolic analysis
    - Their success in vulnerability discovery
    - Challenges in exploitation research
  - Our analysis platform
    - Cross-platform trace generation

- Second Half
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - **Supporting interactive analysis**
    - TREE (Taint-enabled Reverse Engineering Environment)
  - How does TREE and CBASS Interact

Gaps between Research and Analysis

Snapshot of A Popular Security Forum



Status Quo: advanced analysis tools are not available to engineers

Re: Taint tool

by **kareldjag/michk** » Sun Jul 08, 2012 2:24 pm

hi

Unfortunately most tools available are for Linux or Mac; others are Private.
You can find an overview of these tools in this .pdf paper
http://code.google.com/p/tanalysis/downloads/list

rgds

Security? Yeah But Well: http://www.ouaismaisbon.ch/)

Re: Taint tool

by **Cr4sh** » Sun Jul 08, 2012 7:42 pm

Unfotunately, powerful and friendly (for integration with the 3-rd party code analysis software) taint analysis tools are not available for public
Few weeks ago I tried to contact with the autors of dytan framework (to ask for binaries or source code), but got no ansver.

Last edited by **Cr4sh** on Sun Jul 08, 2012 7:46 pm, edited 1 time in total.

http://blog.cr4.sh/

Real World Demands TREE:

Re: Taint tool

by **p4r4n0id** » Sat Jul 27, 2013 6:46 am

Taint-enabled Reverse Engineering Environment (TREE)

https://code.google.com/p/tree-cbass/

Keep Low. Move Fast. Kill First. Die Last. One Shot. One Kill. No Luck. Pure Skill.
http://p4r4n0id.com/

Interactive Binary Analysis

- Automated binary analyses useful for certain tasks (e.g., finding crashes)
- Many binary analyses can’t be automated
- Expert experience and heuristics are still **key** to binary analyses

Interactive Analysis – A Reverse Engineer's Workflow

- A reverse engineer's work is an incremental and iterative process, rather than a straight one
- "An Exploratory Study of Software Reverse Engineering in a Security Context"
  - Binary analysis is at the heart of the project
  - Most of the reverse engineering work is performed using IDA Pro
  - Challenges include complexity and the need to track different levels of details

An Interactive Analysis Process

- Step 1:

  Start from unknown: not sure what you are looking for and what you will find

- Step 2:

  What -if: make a series of hypothesis, prove or disprove them

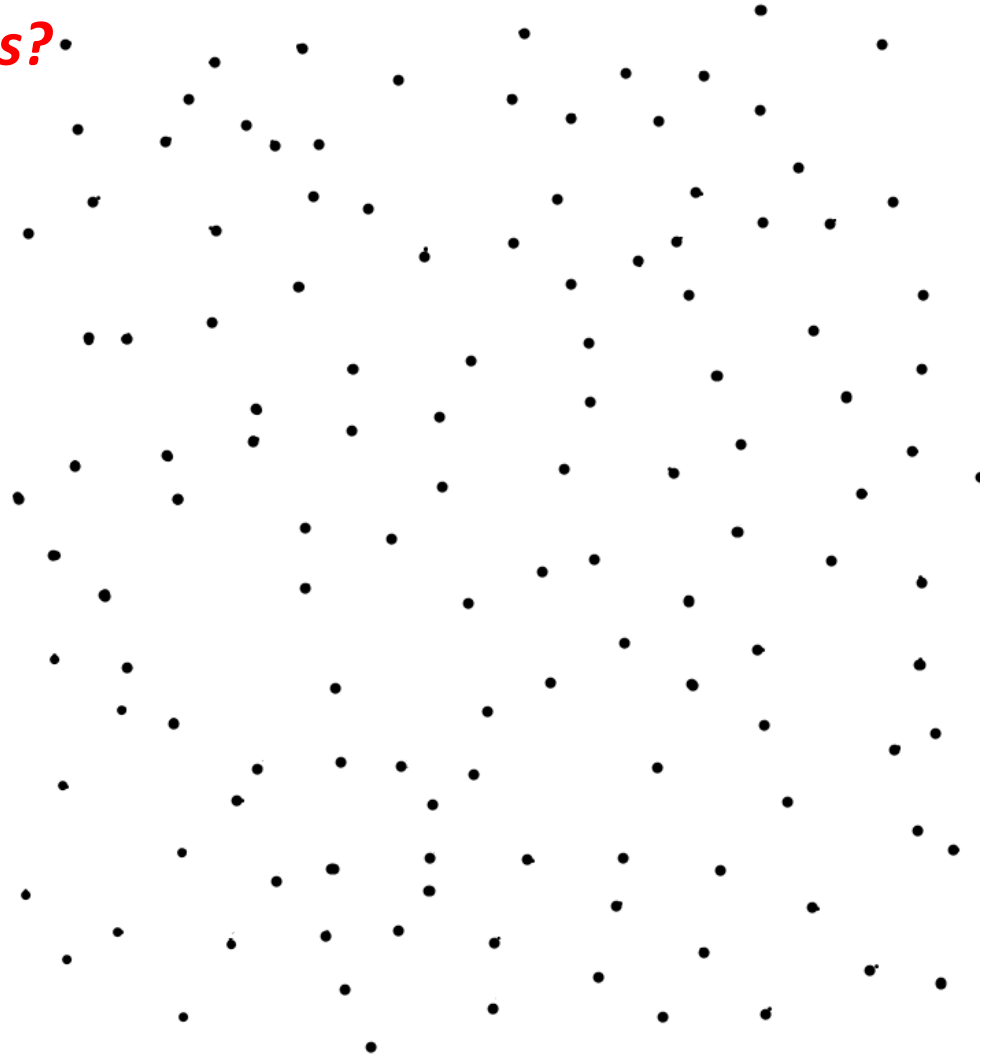
- Step 3:

  Create new hypothesis based on what you have found, if Satisfied or Exhausted Quit

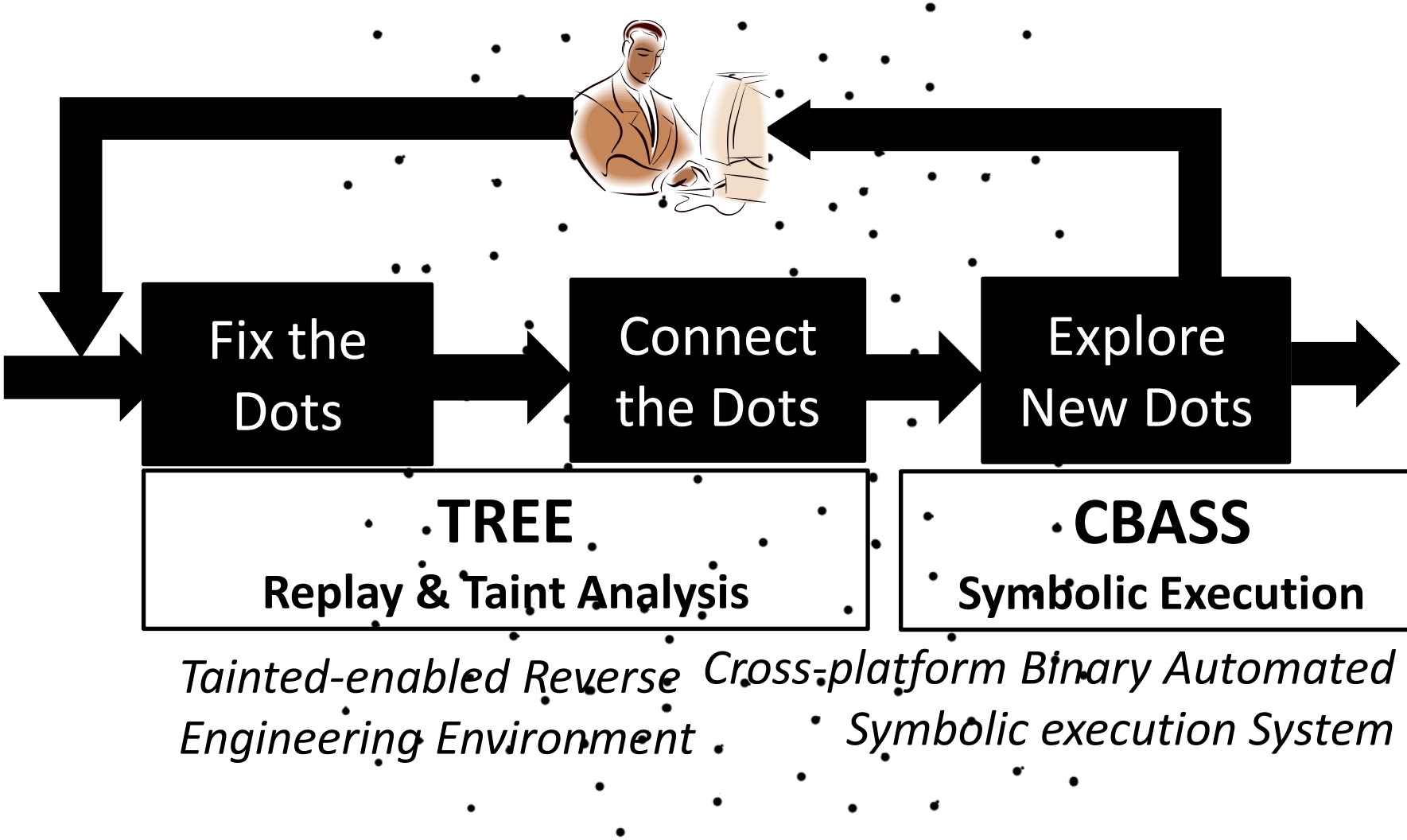
  Else go back to Step 2

Interactive Analysis is Like Connecting Dots

What's in the dots?



Our Tools are Designed to Help Interactive Analysis



Our Tools Support

Fixing the Dots (TREE)

Fix the Dots

- Reverse engineers don't like moving dots
- Why do the dots move?
  - Concurrency (multi-thread/multi-core) brings non-deterministic behavior
- Even for deterministic behavior, reverse engineer needs to analyze the internals repeatedly
  - ASLR guarantees nothing internally will be the same

Fix the Dots

- How does TREE work?
  - Generates the trace at runtime
  - Replays it offline
- TREE trace
  - Captures program state = {Instruction, Thread, Register, Memory}
  - Fully automated generation
- TREE can collect traces from multiple platforms
  - Windows/Linux/Mac OS User/Kernel and real devices (Android/ARM, Cisco routers/MIPS, PowePC)

TREE Taint-based Replay vs. Debug-based Replay

- Debug-replay lets **you connect the dots**
  - Single step, stop at function boundary, Breakpoint
- TREE replay **connects dots for you**
  - Deterministic replay with taint-point break

Our Tools Support

Connecting the Dots (TREE)

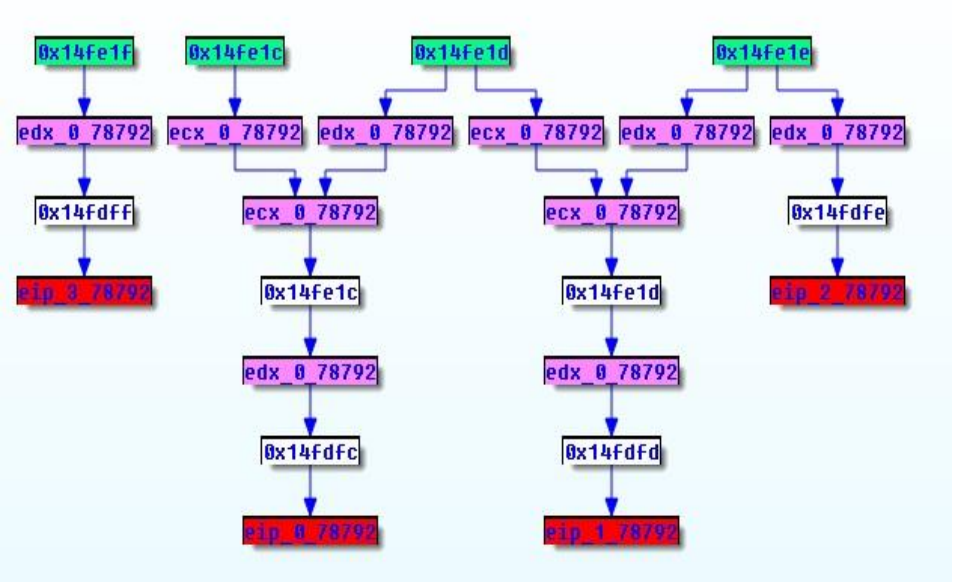
Why connecting the dots?

- Vulnerability root cause needs to determine *precisely* what original, like input, byte (s) impact security target(s) later, and how
- A little enhanced running example:

//INPUT
DWORD dwBytesRead;
ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

//INPUT TRANSFORMATION
for(int i=0; i< (dwBytesRead-2); i++)
sBigBuf[i] +=sBigBuf[i+1];

//PATH CONDITION
If(sBigBuf[0]==‘b’)
//Vulnerable Function
StackOverflow(sBigBuf,dwBytesRead);



Connecting Dots is Hard

- Basic elements are complex in real programs
  - Code size can be thousands (++) of lines
  - Inputs can come from many places
  - Transformations can be lengthy
  - Paths grow exponentially
- Basic elements are likely separated by millions of instructions, spatially and temporally
- Multiple protections built in

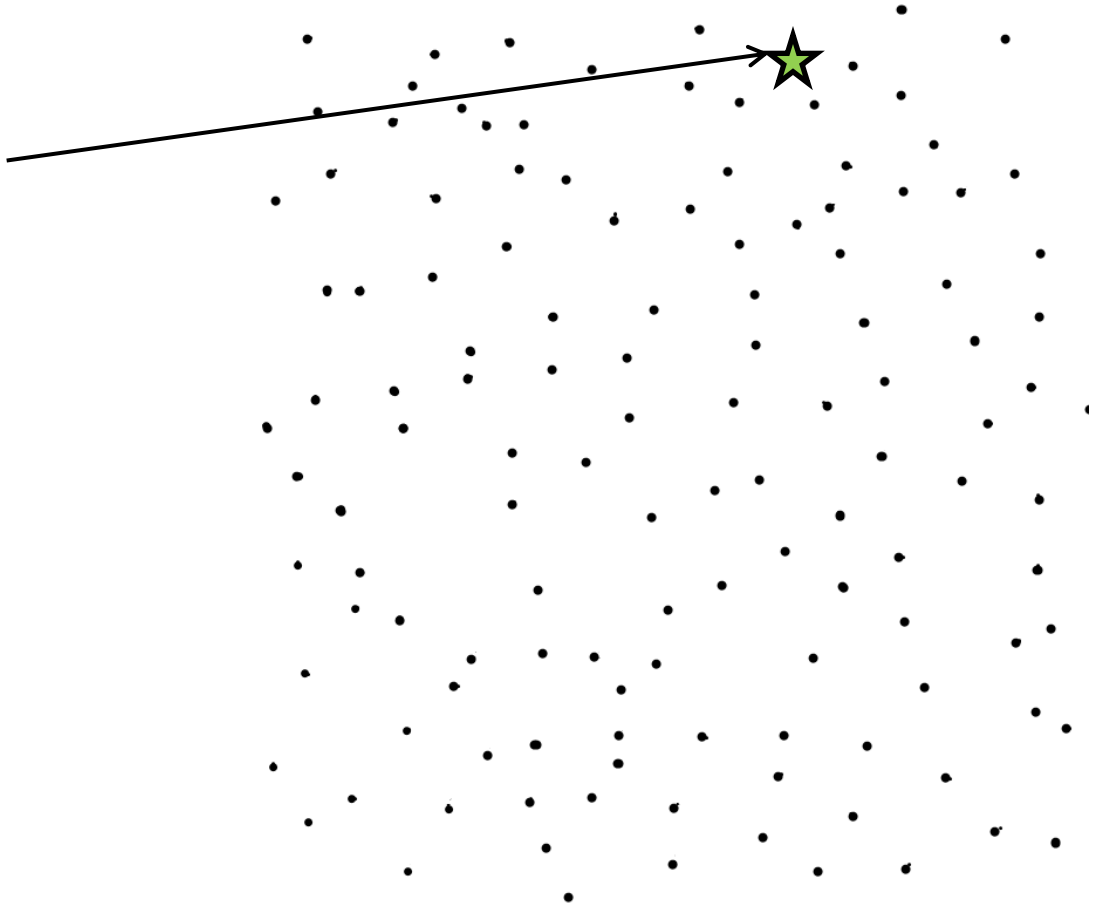
Techniques Help Connect the Dots

- Dynamic Taint Analysis
  - Basic Definitions
    - Taint source
    - Taint Sink:
    - Taint Policy:
- Taint-based Dynamic Slicing
  - Taint focused on data
  - Slicing focused on relevant instructions and sequences

Connect the Dots

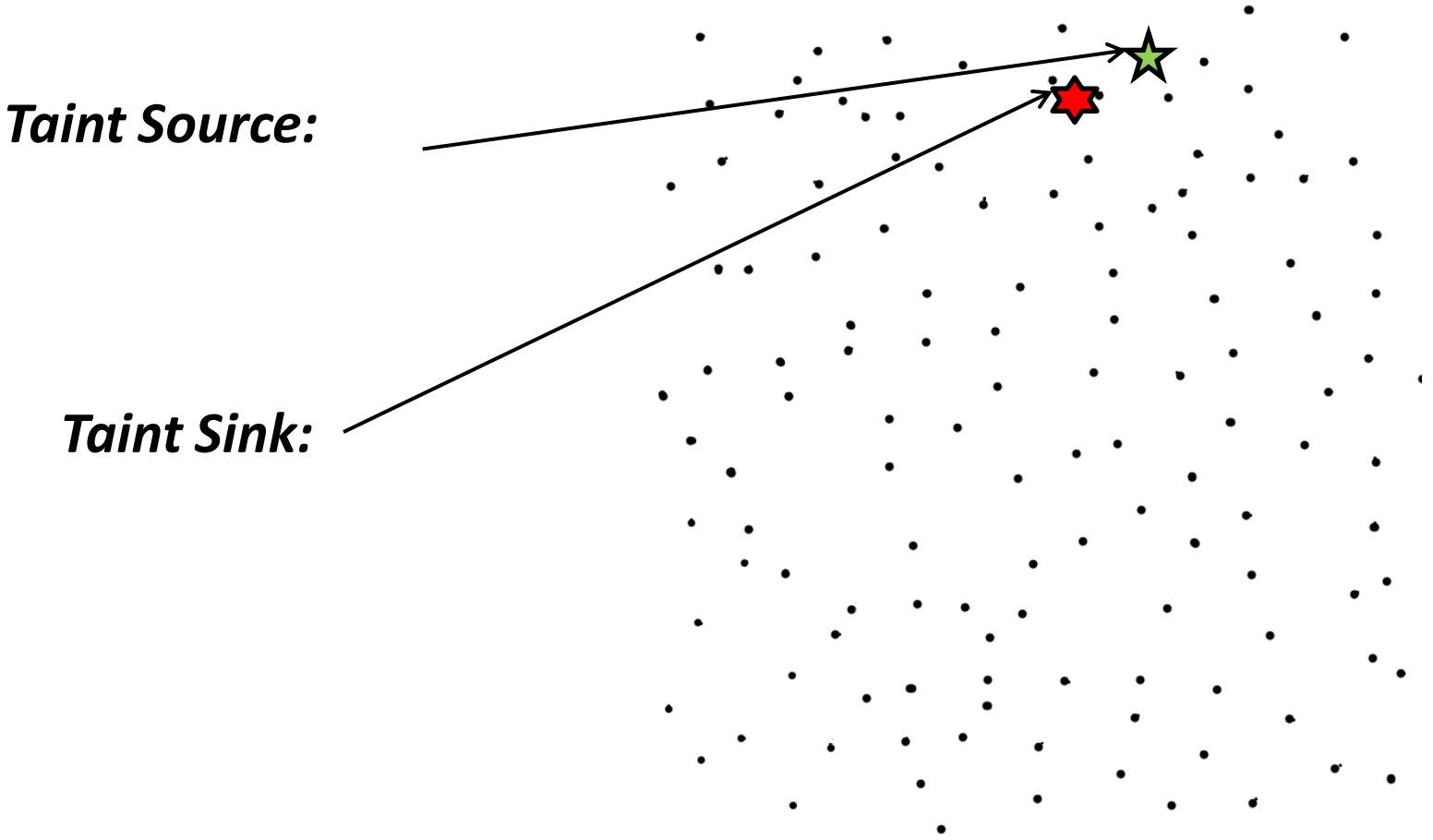
- TREE connects dots -- using taint analysis

Taint Source:



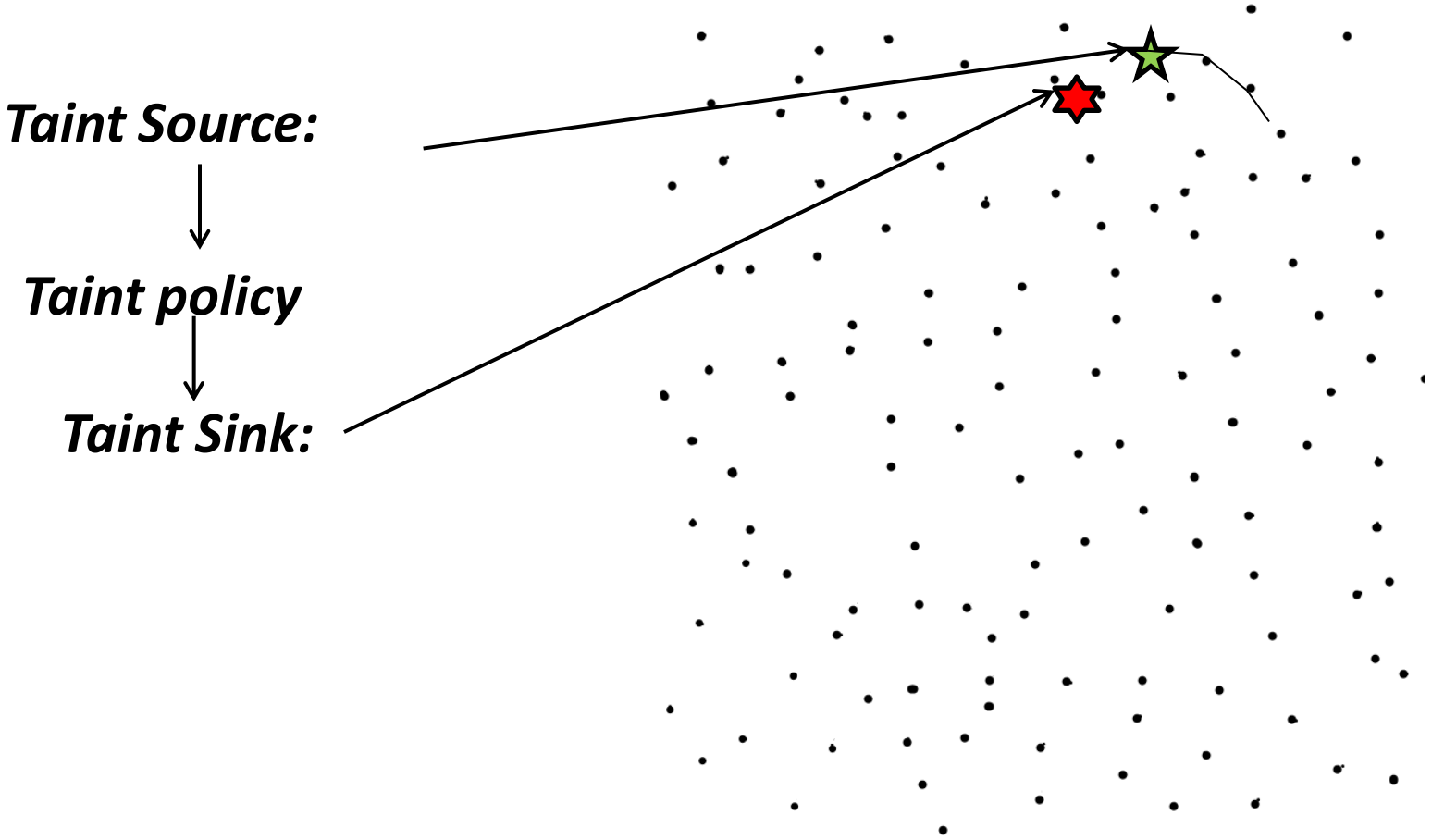
Connect the Dots

- TREE connects dots -- using taint analysis



Connect the Dots

- TREE connects dots -- using taint analysis



Connect the Dots

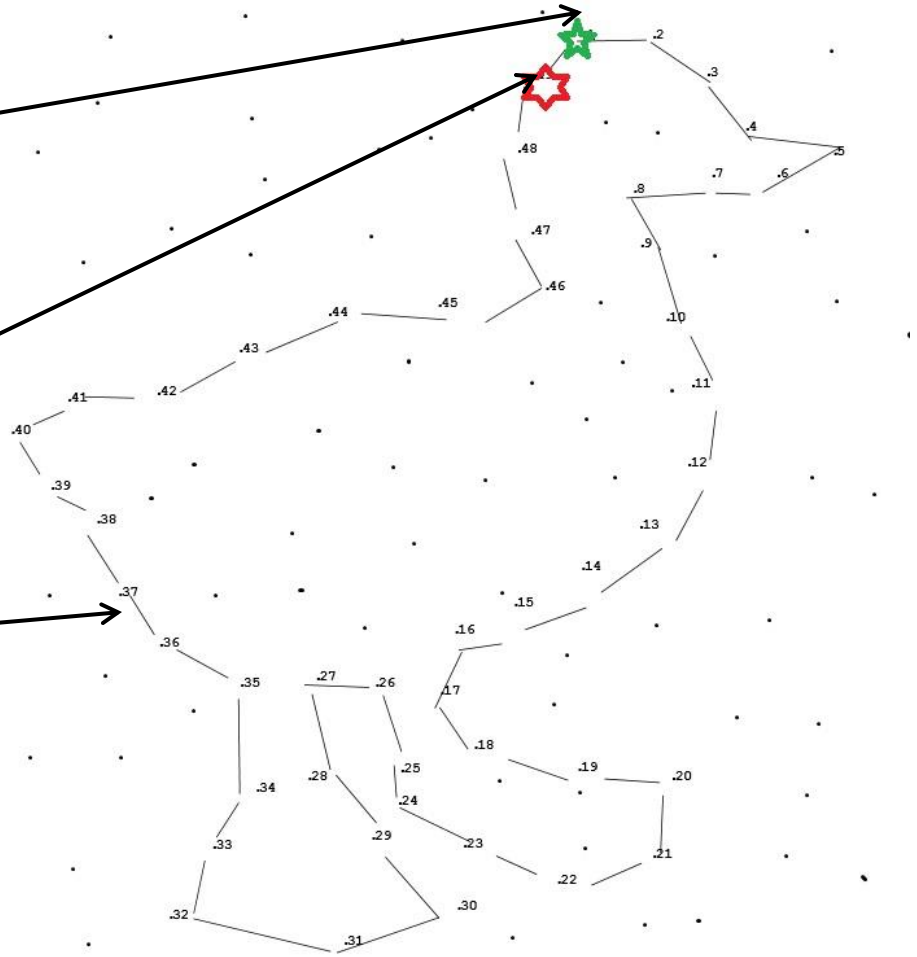
- TREE connects dots -- using taint analysis

Taint Source:

Taint policy

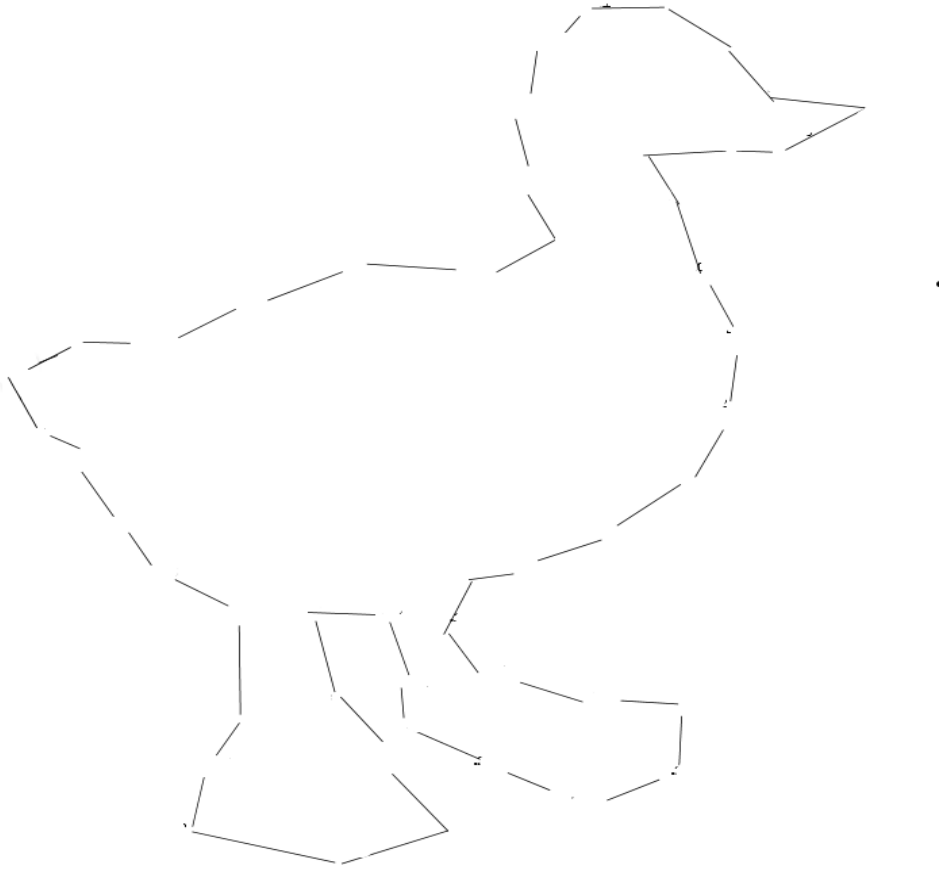
Taint Sink:

- **Dynamic Slicing**



Find the Dots and Slice that Matter

In practice, most dots don’t matter – eliminate them quickly to focus on what matters



Connecting Dots in Running Example

Taint Source: (Input)

↓

Taint policy (Data)

↓

Taint Sink: eip

The Slice

call ds:ReadFile

↓

movb (%eax), %dl

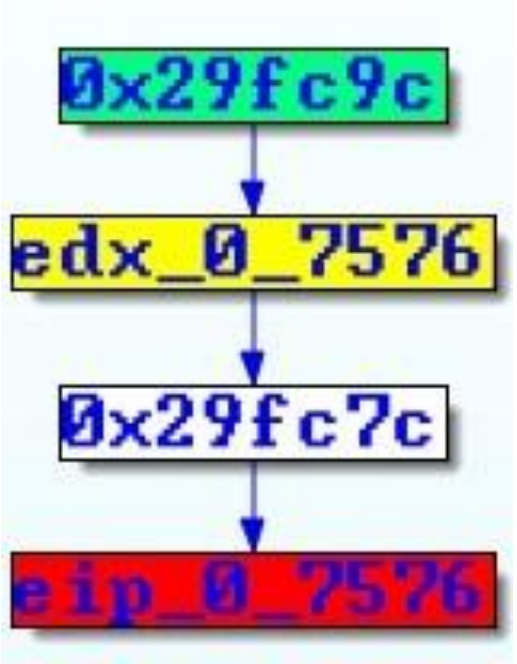
↓

movb %dl, -0x8(%ebp,%ecx,1)

↓

retl

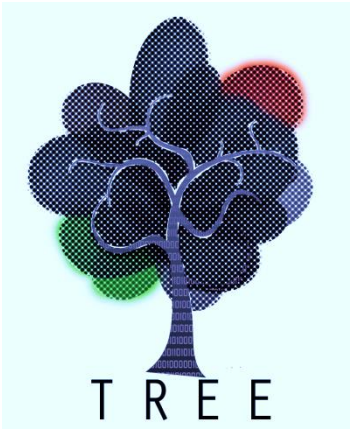
The Taint Graph



What You Connect is What You Get

- **ABCD** – Connect Dots in different ways
  - **A**ddress dependency
  - **B**ranch conditions
  - **C**ounter (Loop)
  - **D**ata dependency

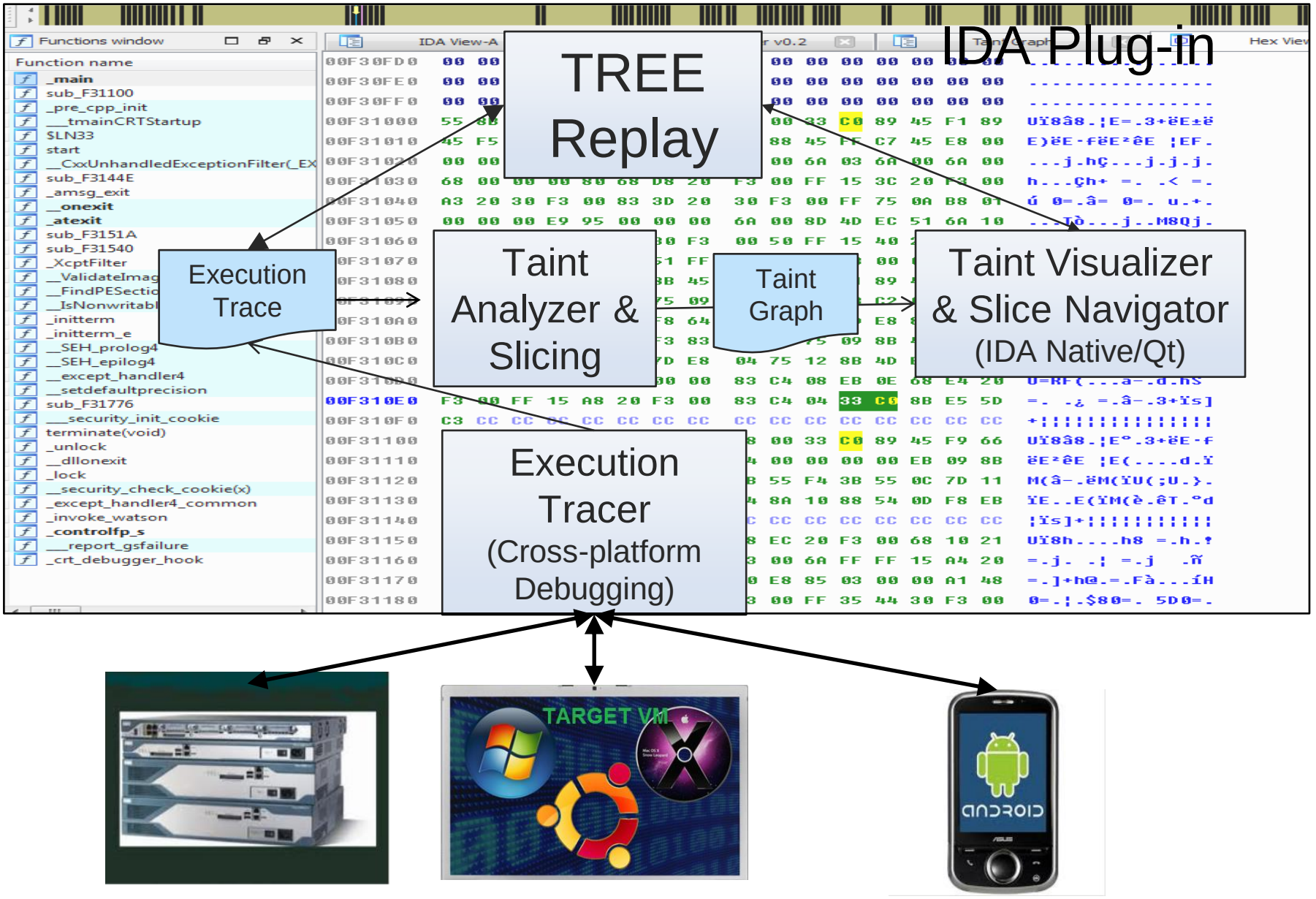
- Each dependency (taint policy) has different security implications



TREE

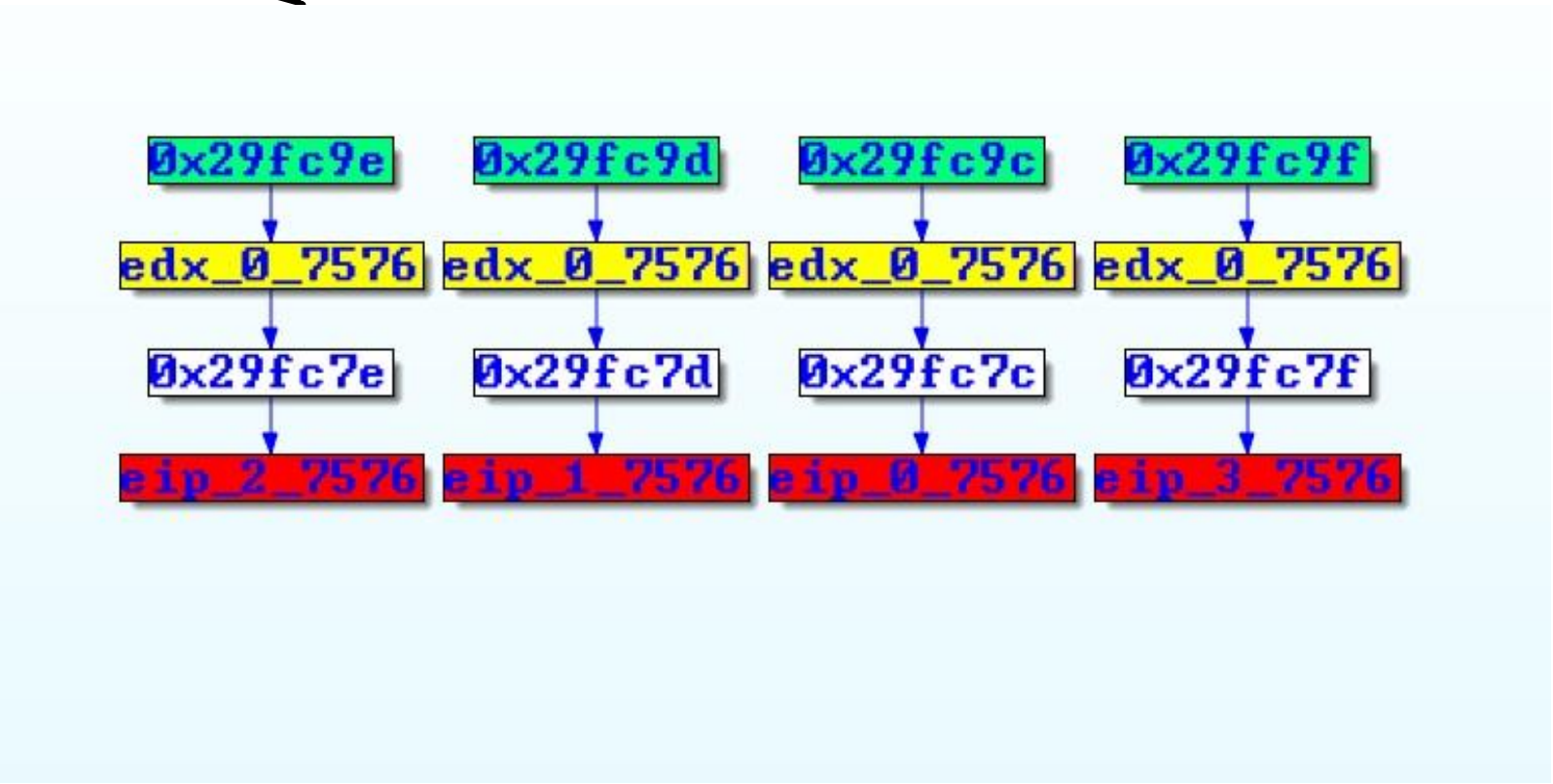
TAINT-ENABLED REVERSE ENGINEERING ENVIRONMENT

TREE Key Components



TREE: The Front-end of Our Interactive Analysis System

Taint Graph



TREE: The Front-end of Our Interactive Analysis System

Taint Table

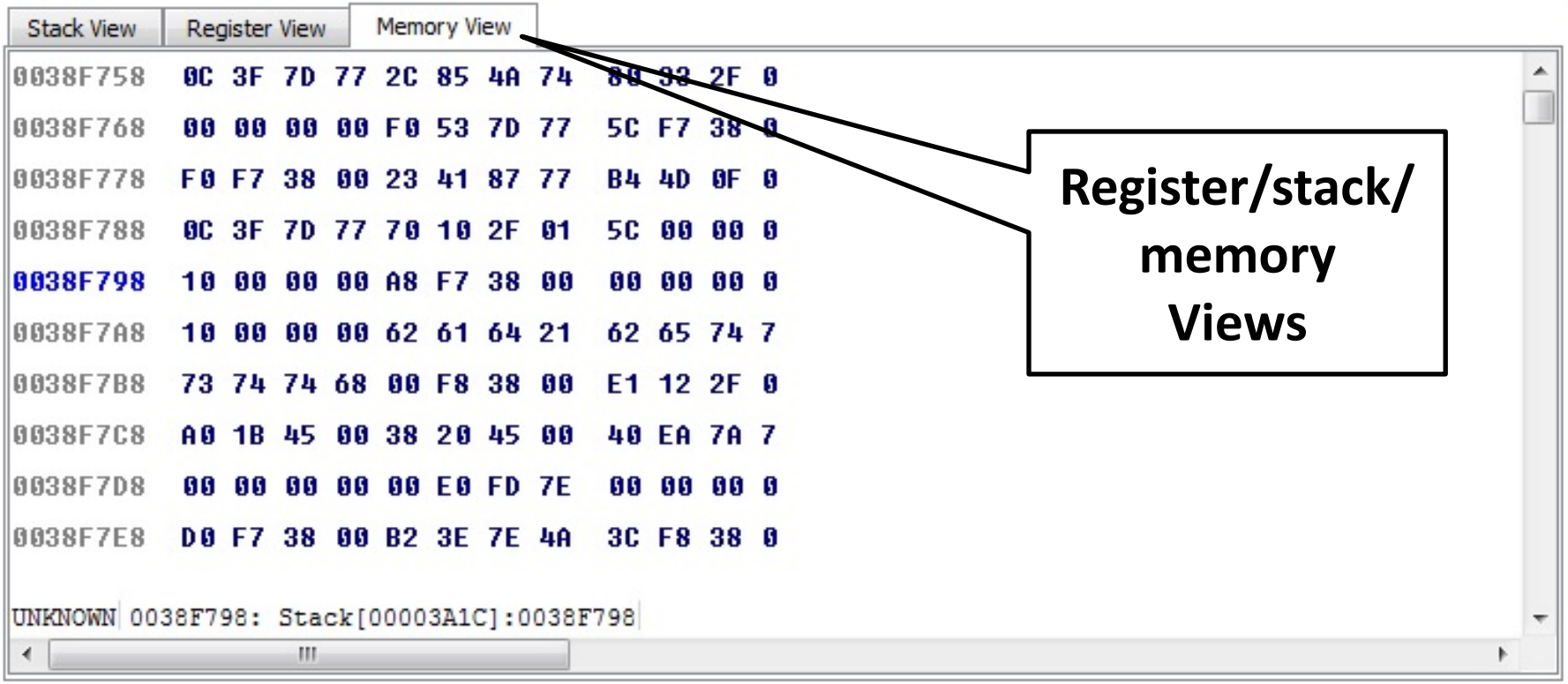
UUID	Type	Name	Start Sequence	End Sequence	formation Instru	Child C	Child D
60	register	eip_3_14876	0x11c		retl		52
59	register	eip_2_14876	0x11c		retl		50
58	register	eip_1_14876	0x11c		retl		48
57	register	eip_0_14876	0x11c		retl		46
52	memory	0x38f79b	0x112		movb %dl, -...		51
51	register	edx_0_14876	0x111	0x117	movb (%eax), %dl		16
50	memory	0x38f79a	0x106		movb %dl, -...		49
49	register	edx_0_14876	0x105	0x10b	movb (%eax)...		15
48	memory	0x38f799	0xfa		movb %dl, -...		47
47	register	edx_0_14876	0xf9	0xff	movb (%eax)...		14
46	memory	0x38f798	0xee		movb %dl, -0x8(%ebp,%...		45
45	register	edx_0_14876	0xed	0xf3	movb (%eax), %dl		13
16	input	0x38f7bb	0x0		0x12f106a		

TREE: The Front-end of Our Interactive Analysis System

Execution Trace Table

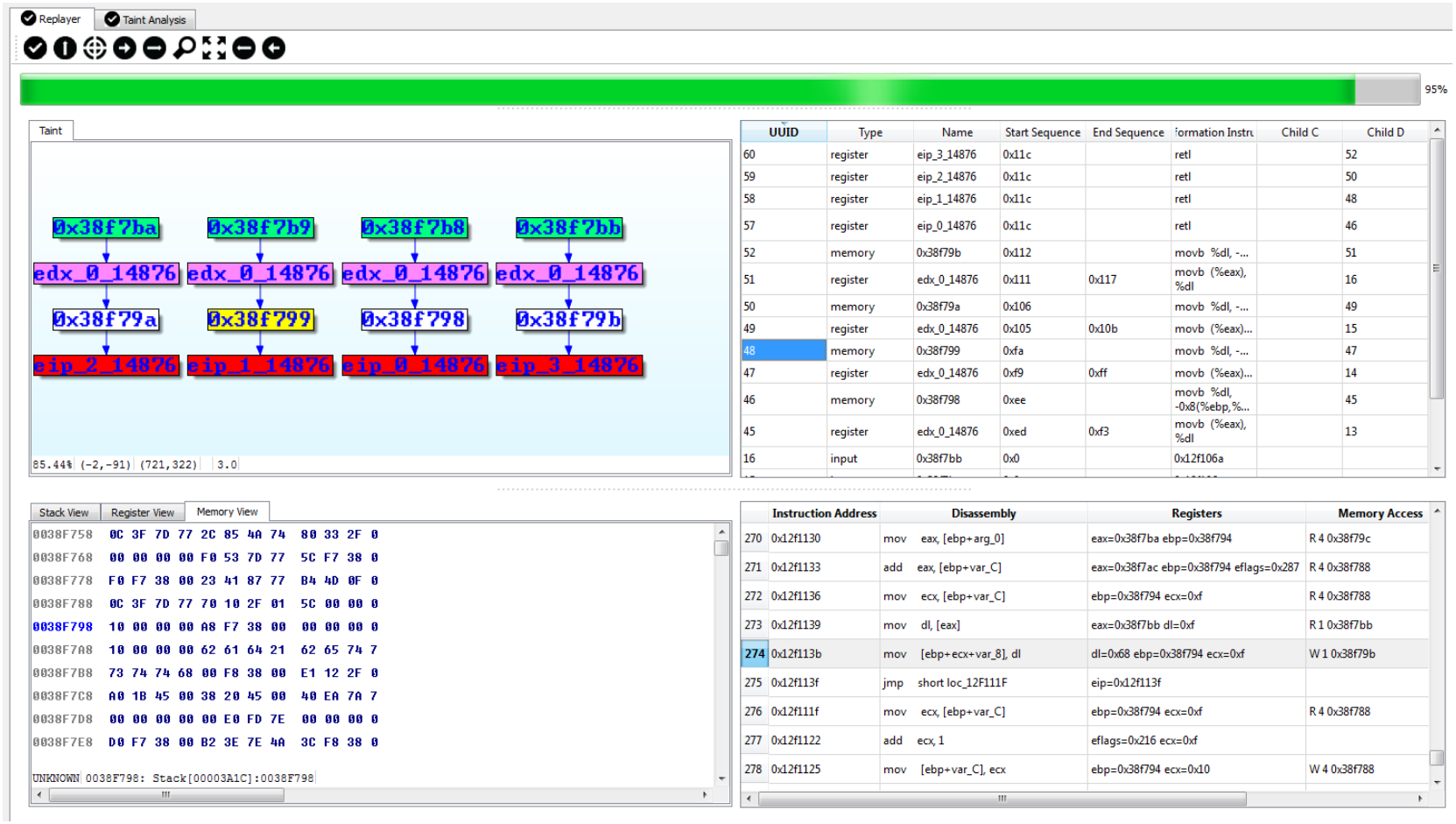
	Instruction Address	Disassembly	Registers	Memory Access
270	0x12f1130	mov eax, [ebp+arg_0]	eax=0x38f7ba ebp=0x38f794	R 4 0x38f79c
271	0x12f1133	add eax, [ebp+var_C]	eax=0x38f7ac ebp=0x38f794 eflags=0x287	R 4 0x38f788
272	0x12f1136	mov ecx, [ebp+var_C]	ebp=0x38f794 ecx=0xf	R 4 0x38f788
273	0x12f1139	mov dl, [eax]	eax=0x38f7bb dl=0xf	R 1 0x38f7bb
274	0x12f113b	mov [ebp+ecx+var_8], dl	dl=0x68 ebp=0x38f794 ecx=0xf	W 1 0x38f79b
275	0x12f113f	jmp short loc_12F111F	eip=0x12f113f	
276	0x12f111f	mov ecx, [ebp+var_C]	ebp=0x38f794 ecx=0xf	R 4 0x38f788
277	0x12f1122	add ecx, 1	eflags=0x216 ecx=0xf	
278	0x12f1125	mov [ebp+var_C], ecx	ebp=0x38f794 ecx=0x10	W 4 0x38f788

TREE: The Front-end of Our Interactive Analysis System



TREE: The Front-end of Our Interactive Analysis System

Replay is focal point of user interaction



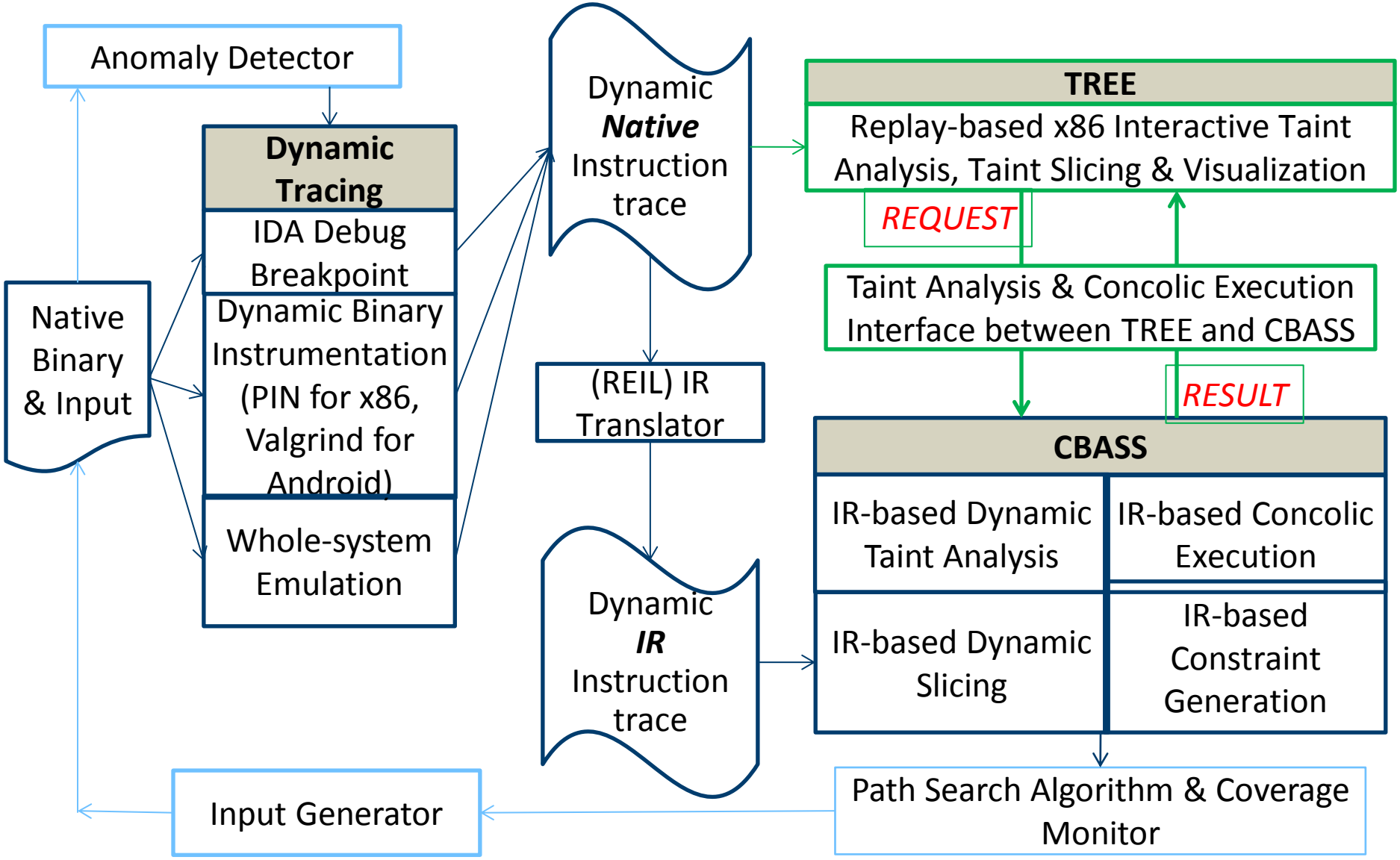
Tree Demo

Using TREE to Analyze a Crash

Outline

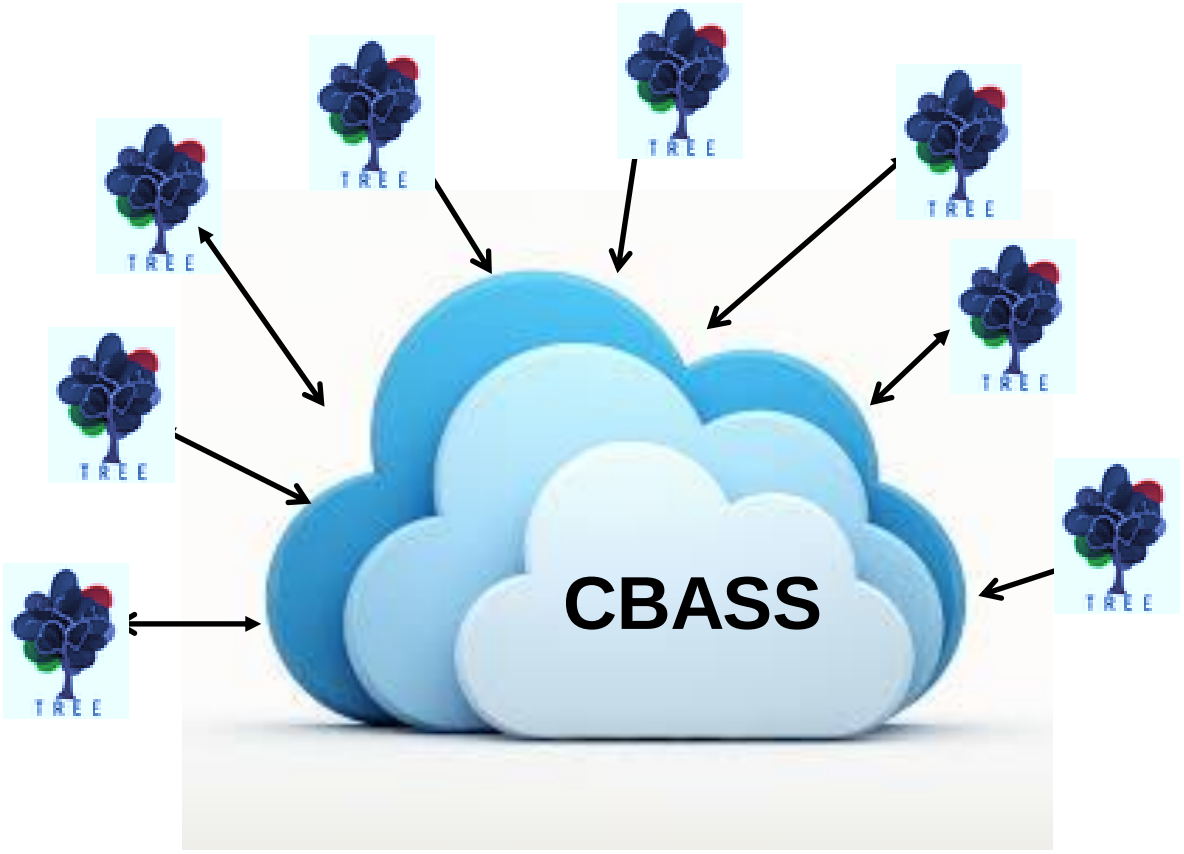
- First Half
  - Background
    - Dynamic analysis and symbolic analysis
    - Their success in vulnerability discovery
    - Challenges in exploitation research
  - Our analysis platform
    - Cross-platform trace generation
- Second Half
  - Supporting automated analysis
    - CBASS (Cross-platform Symbolic-execution System)
  - Supporting interactive analysis
    - TREE (Taint-enabled Reverse Engineering Environment)
  - **How does TREE and CBASS Interact**

Integrate Automated and Interactive Analysis System around CBASS

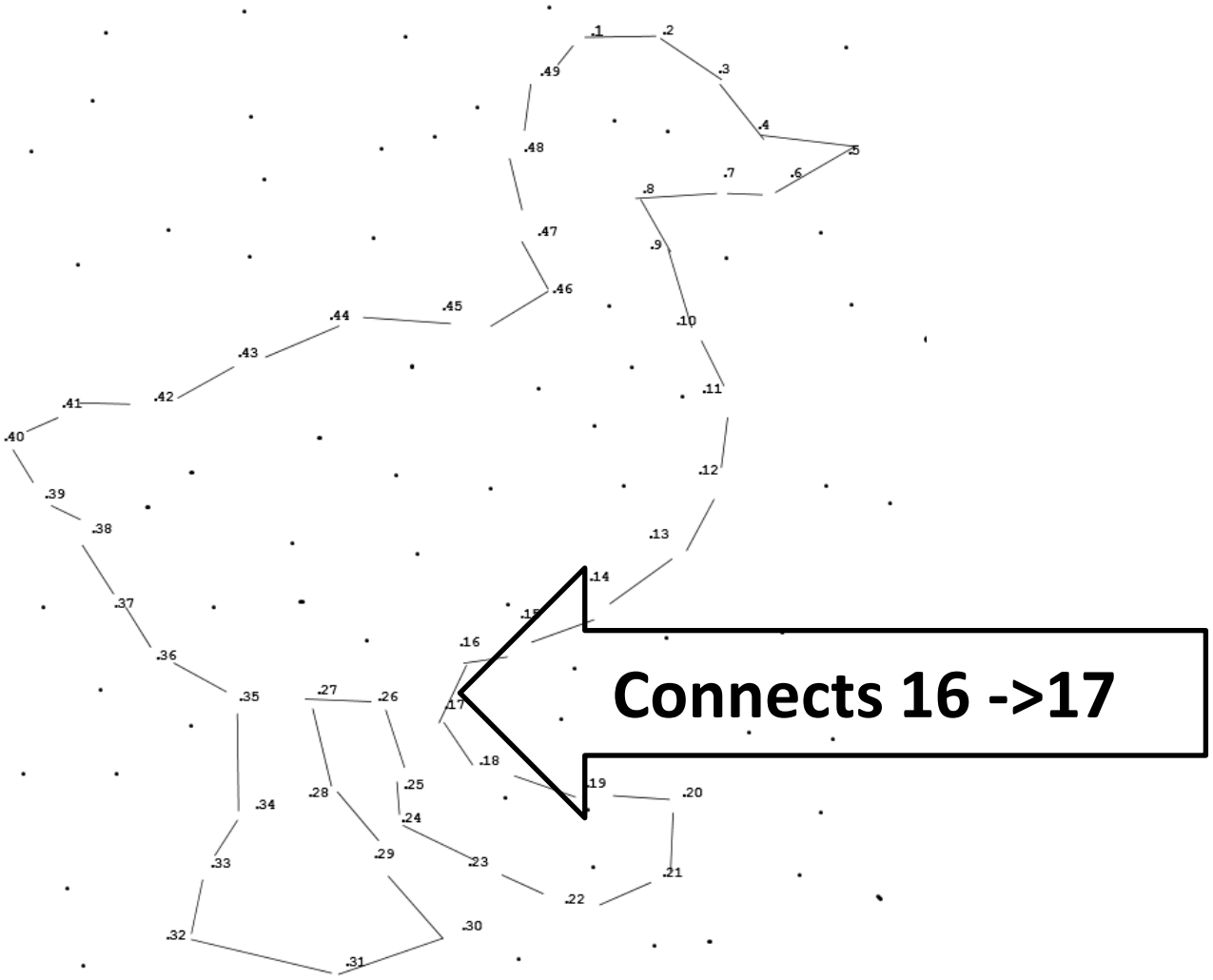


TREE and CBASS in the Future:

- Users use TREE to access service
- CBASS runs in parallel and can be deployed in Cloud

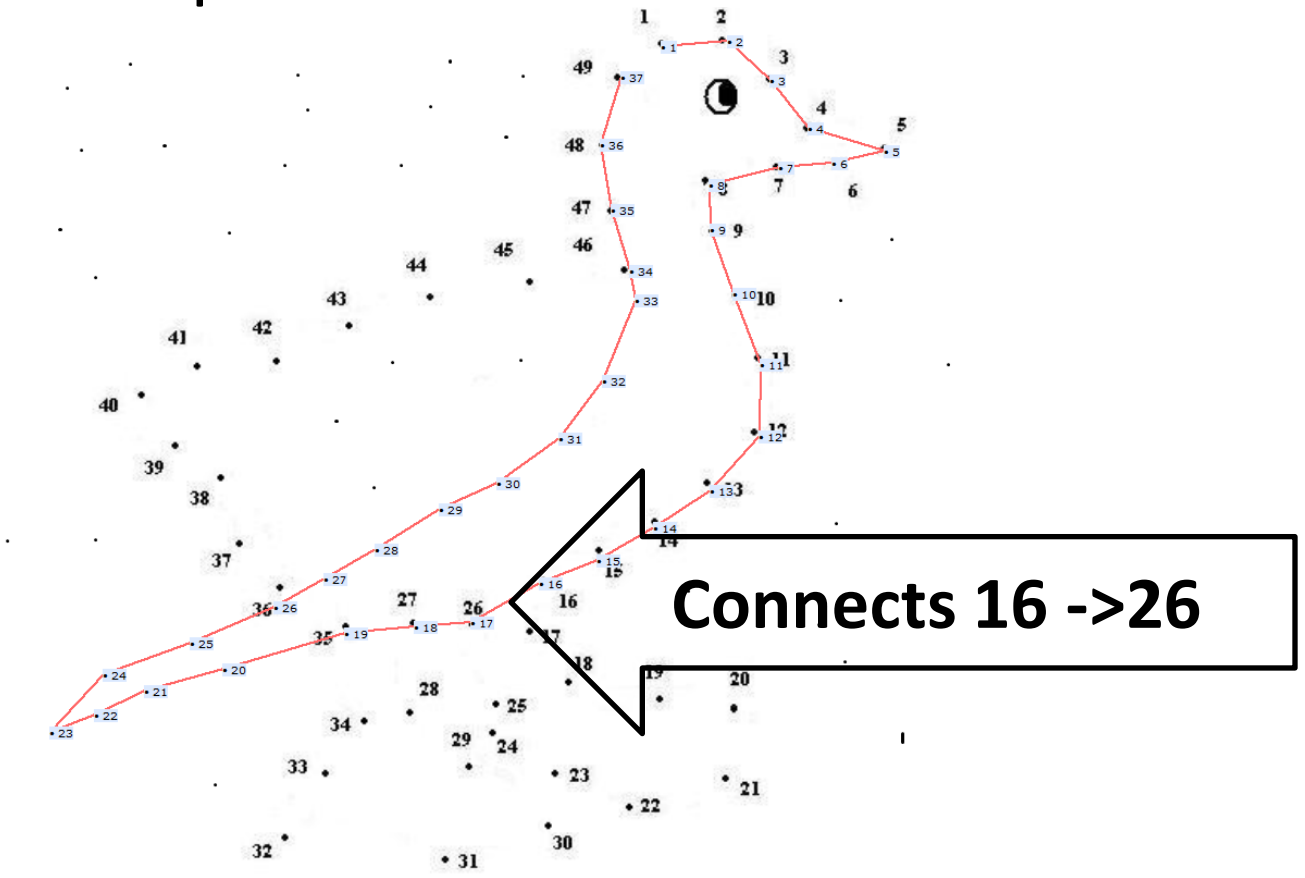


A Key Branch Point for a Duck



The Path for a …

- Reverse engineers don't just connect dots; they want to explore new dots:



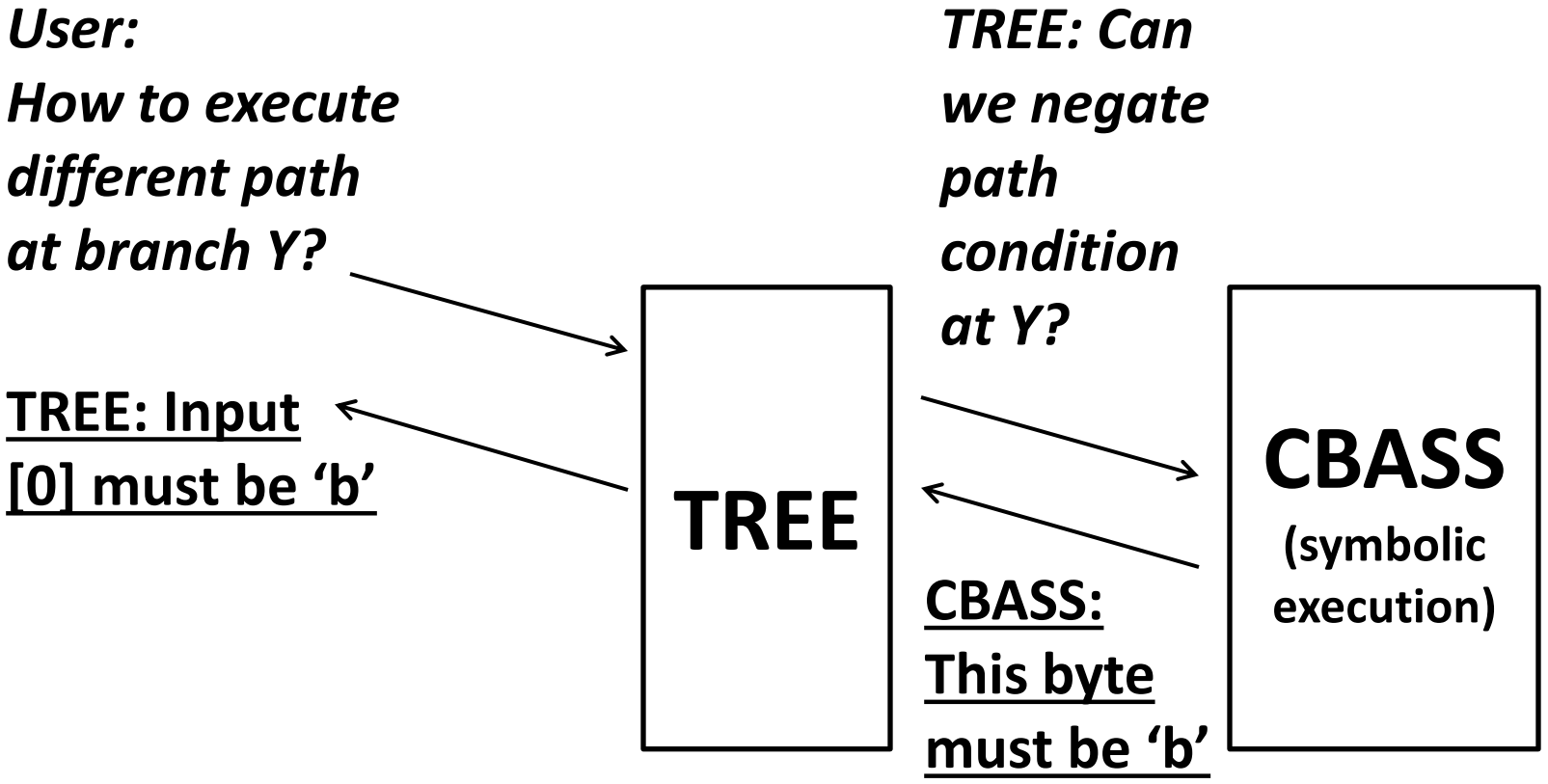
Explore New Dots

- How do you force the program to take a different path to lead to “bad!”?

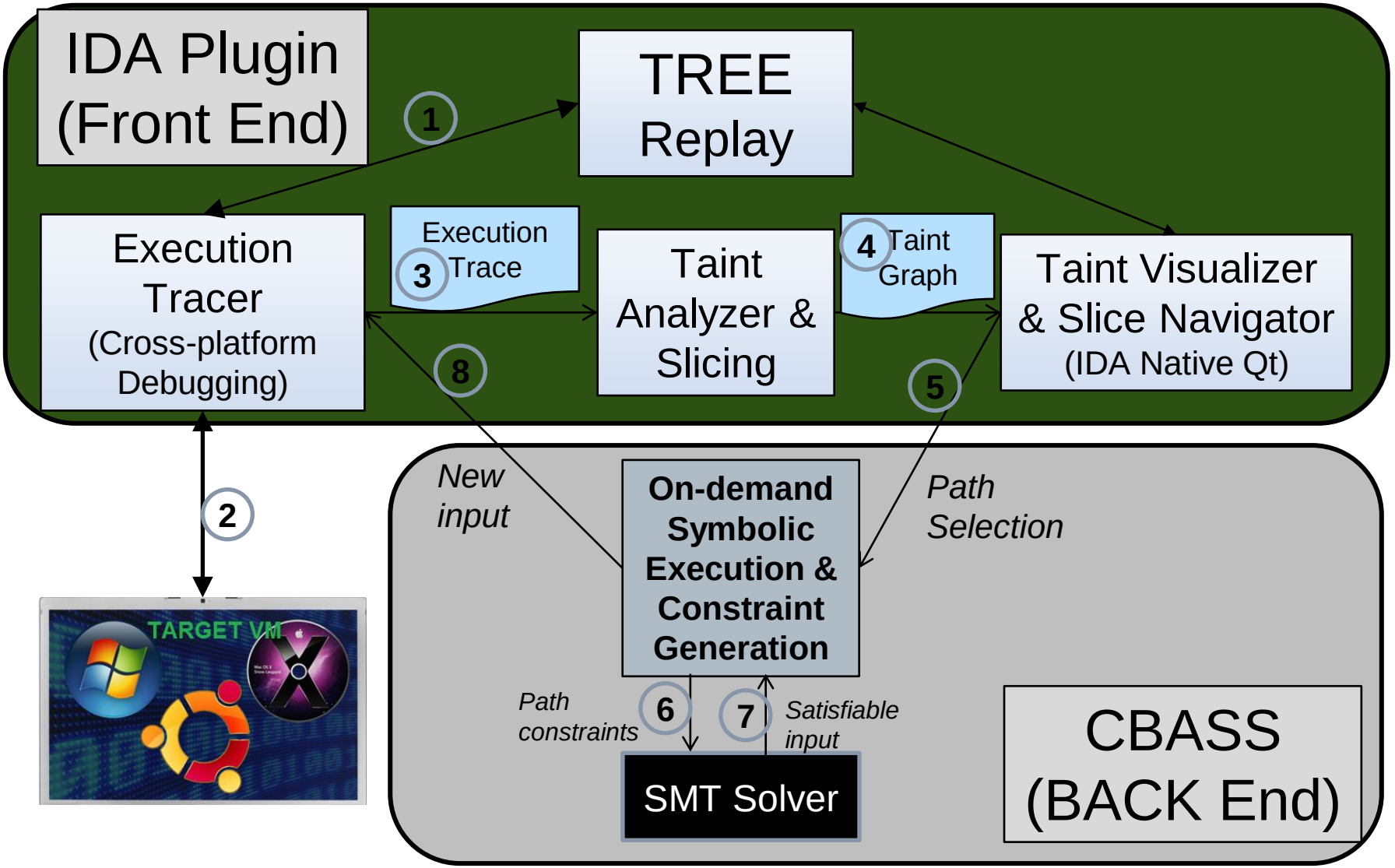
```
//INPUT
ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);
……
//PATH CONDITION
if(sBigBuf[0]=='b') iCount++;
if(sBigBuf[1]=='a') iCount++;
if(sBigBuf[2]=='d') iCount++;
if(sBigBuf[3]=='!') iCount++;
if(iCount==4) // “bad!” path
        StackOVflow(sBigBuf,dwBytesRead) ?
Else // “Good” path
        printf(“Good!”);
```

Explore New Dots

- User wants execution to take different path at a branch point Y – what input will make that happen?



Explore New Dots Demo



Task 1:
Force the Program to Take “bad!” Path

***//*INPUT**

ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

***//*INPUT TRANSFORMATION**

……

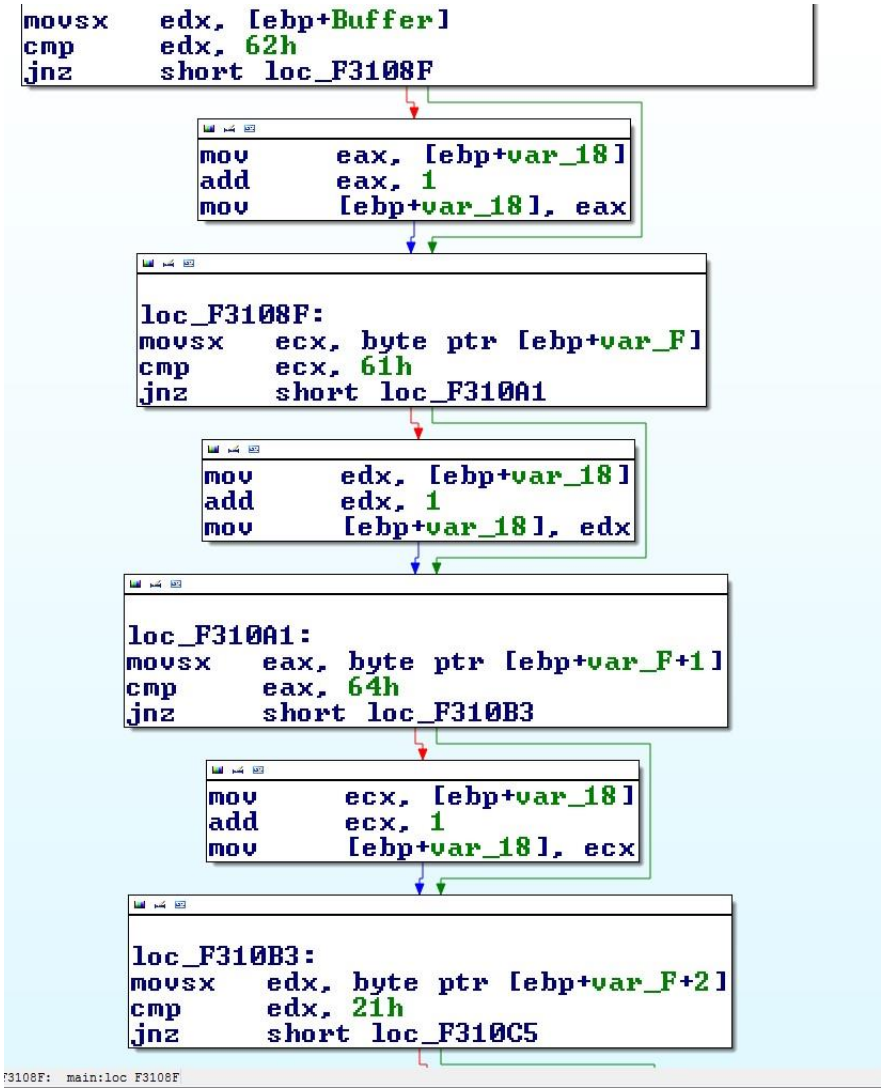
***//*PATH CONDITION**

if(sBigBuf[0]=='b') iCount++;
if(sBigBuf[1]=='a') iCount++;
if(sBigBuf[2]=='d') iCount++;
if(sBigBuf[3]=='!') iCount++;
if(iCount==4) ***// “bad!” path***

***//*Vulnerable Function**

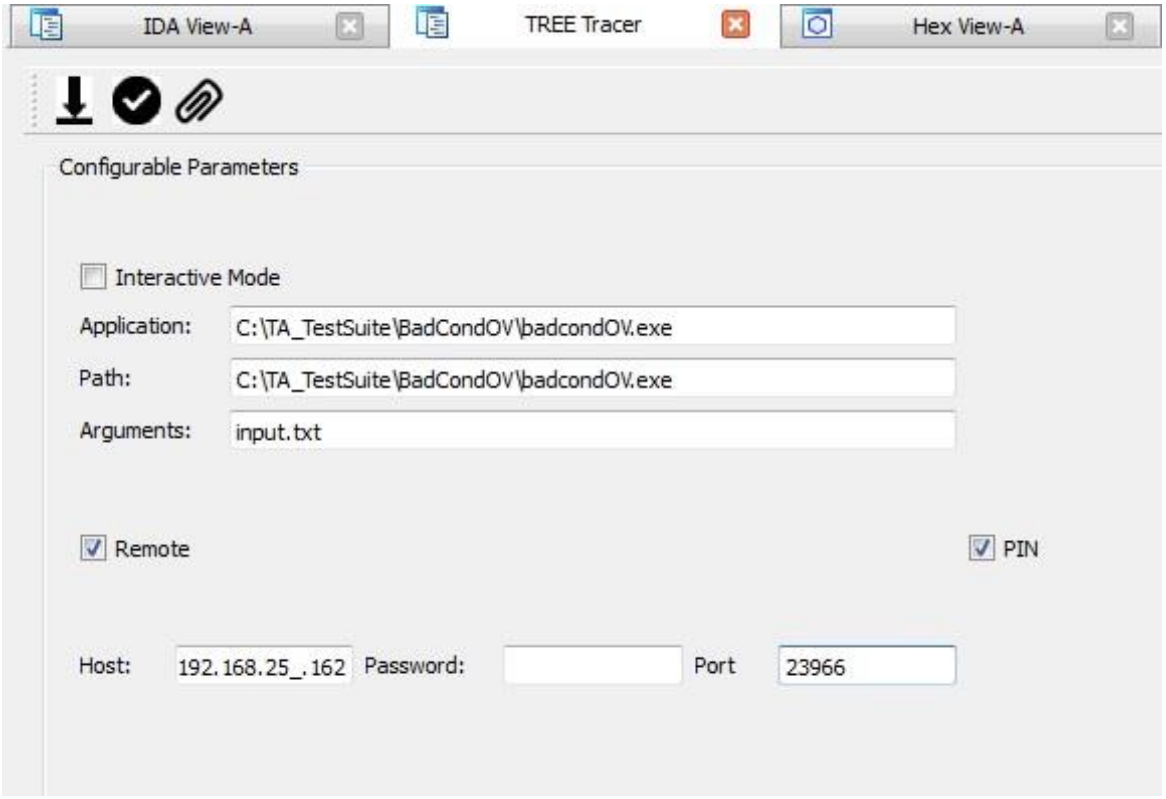
StackOVflow(sBigBuf,dwBytesRead)
else
printf(“Good!”);

Branch Conditions In Disassembly



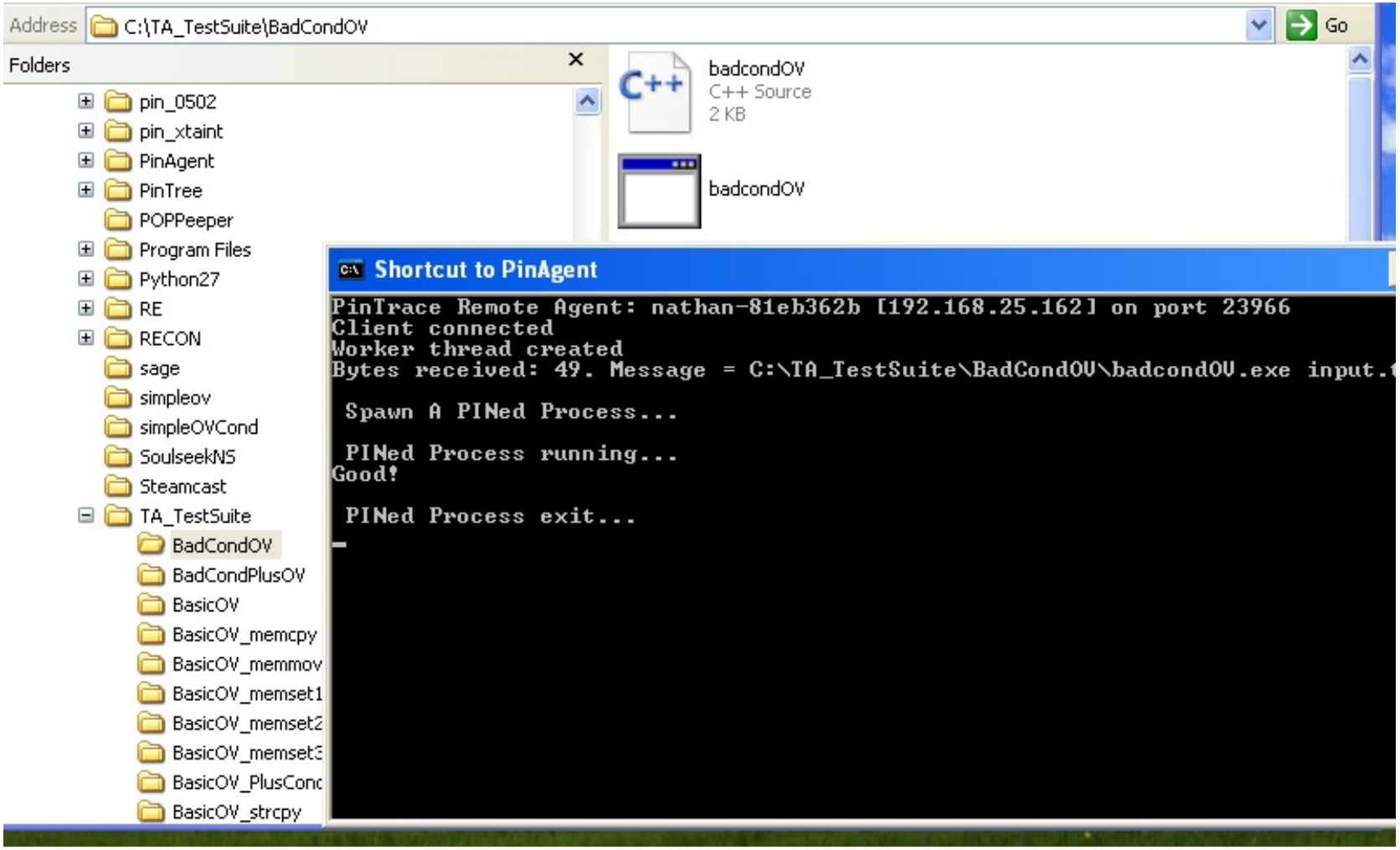
① TREE Pin Trace

PIN: A popular Dynamic Binary Instrumentation (DBI) Framework
http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

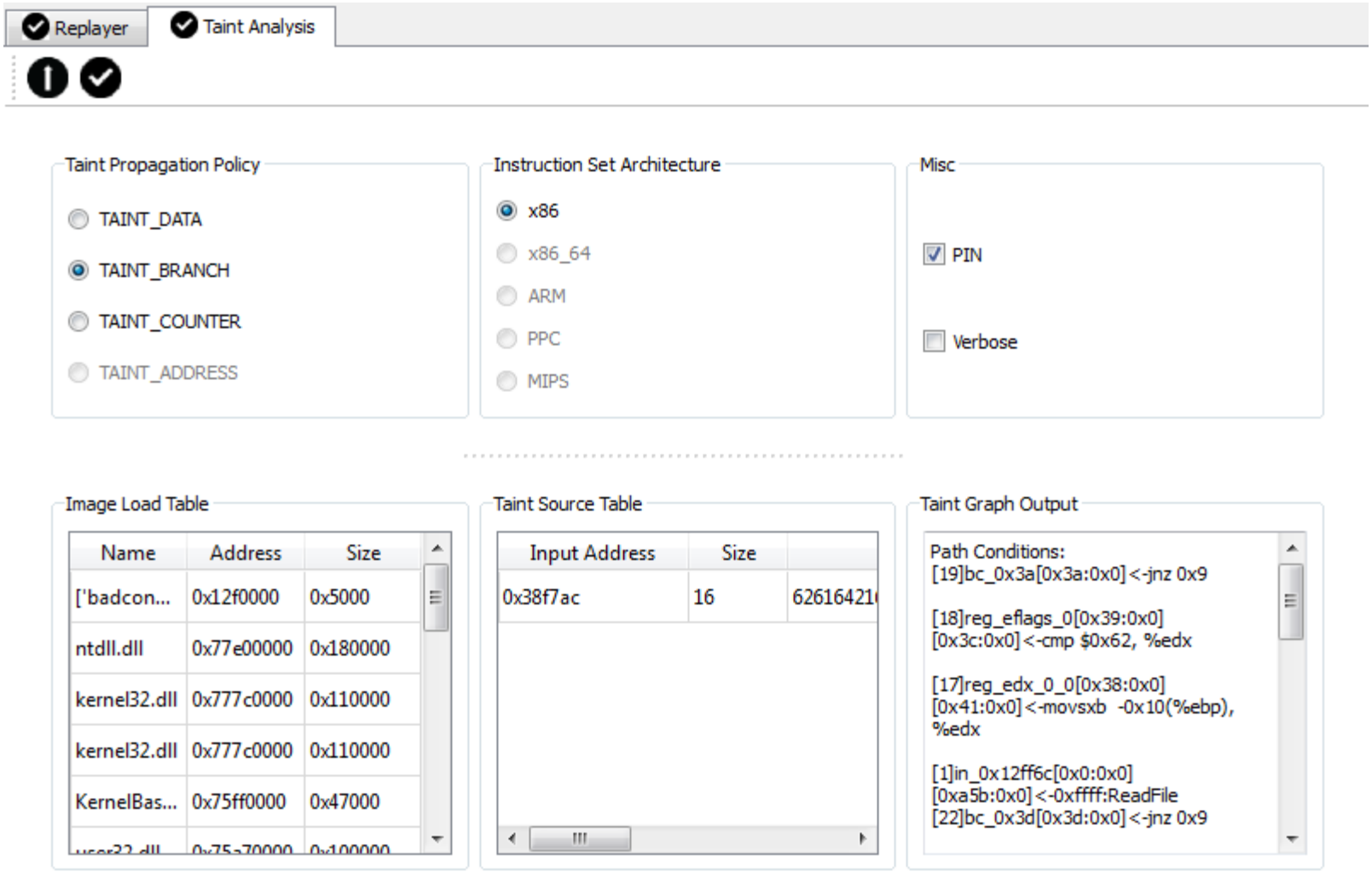


②TREE Console: Trace Generation

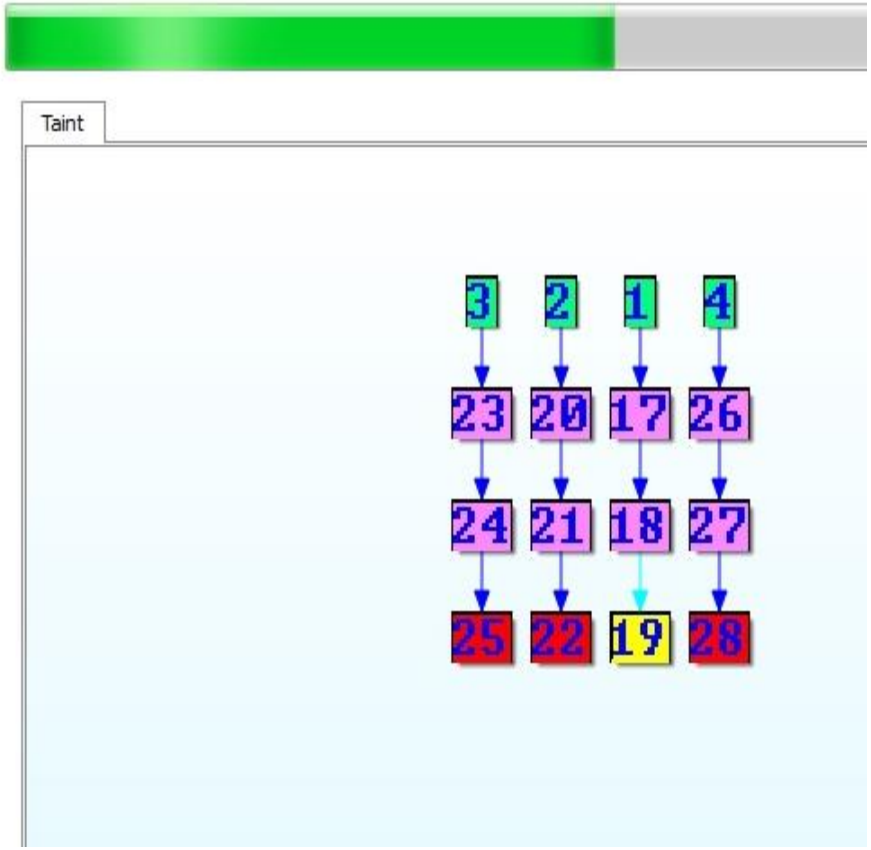
PINAgent: Connects TREE with PIN tracer



③ TREE: Taint Analysis Configuration

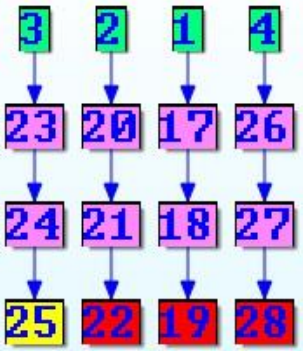


④ TREE: Branch Taint Graph



UUID	Type	Name
1	input	0x12ff6c
17	register	edx_0_0
18	register	eflags_0
19	branch	0x3a
2	input	0x12ff6d
20	register	ecx_0_0
21	register	eflags_0
22	branch	0x3d
23	register	eax_0_0
24	register	eflags_0
25	branch	0x40
26	register	edx_0_0
27	register	eflags_0
28	branch	0x43
3	input	0x12ff6e
4	input	0x12ff6f

⑤Negate Tainted Path Condition to Exercise a New (“Bad”) Path



“Bad!” Path Query

CBASS (Cross-platform Symbolic Execution)

Result

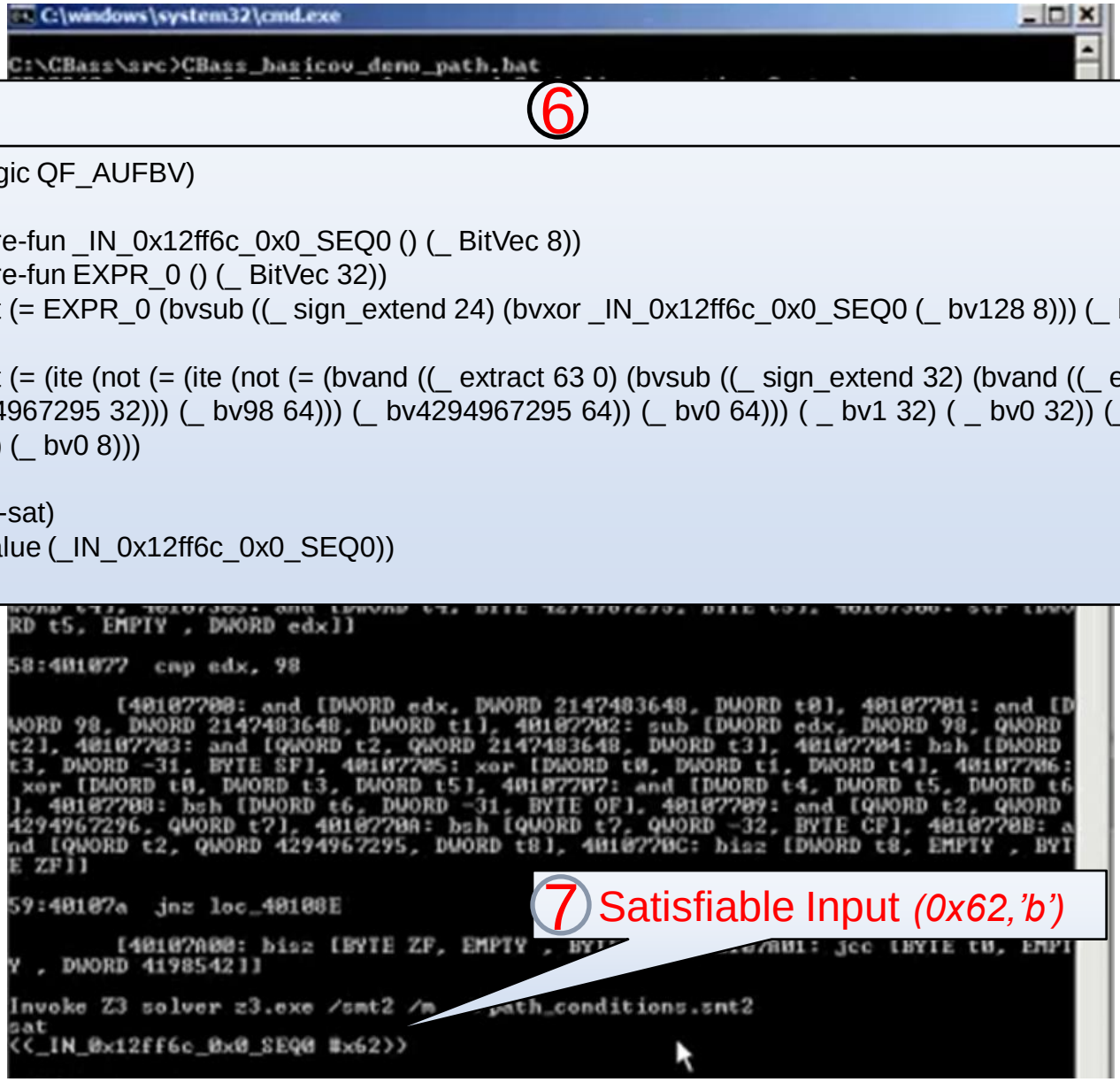
```
Connecting to CBASS server at 127.0.0.1:8888
Query on branch condition of: [19]bc_0x3a
[jnz 0x9]
Result: Offset=0,Value=98
Connecting to CBASS server at 127.0.0.1:8888
Query on branch condition of: [22]bc_0x3d
[jnz 0x9]
Result: Offset=1,Value=97
Connecting to CBASS server at 127.0.0.1:8888
Query on branch condition of: [25]bc_0x40
[jnz 0x9]
Result: Offset=2,Value=100
```

‘b’

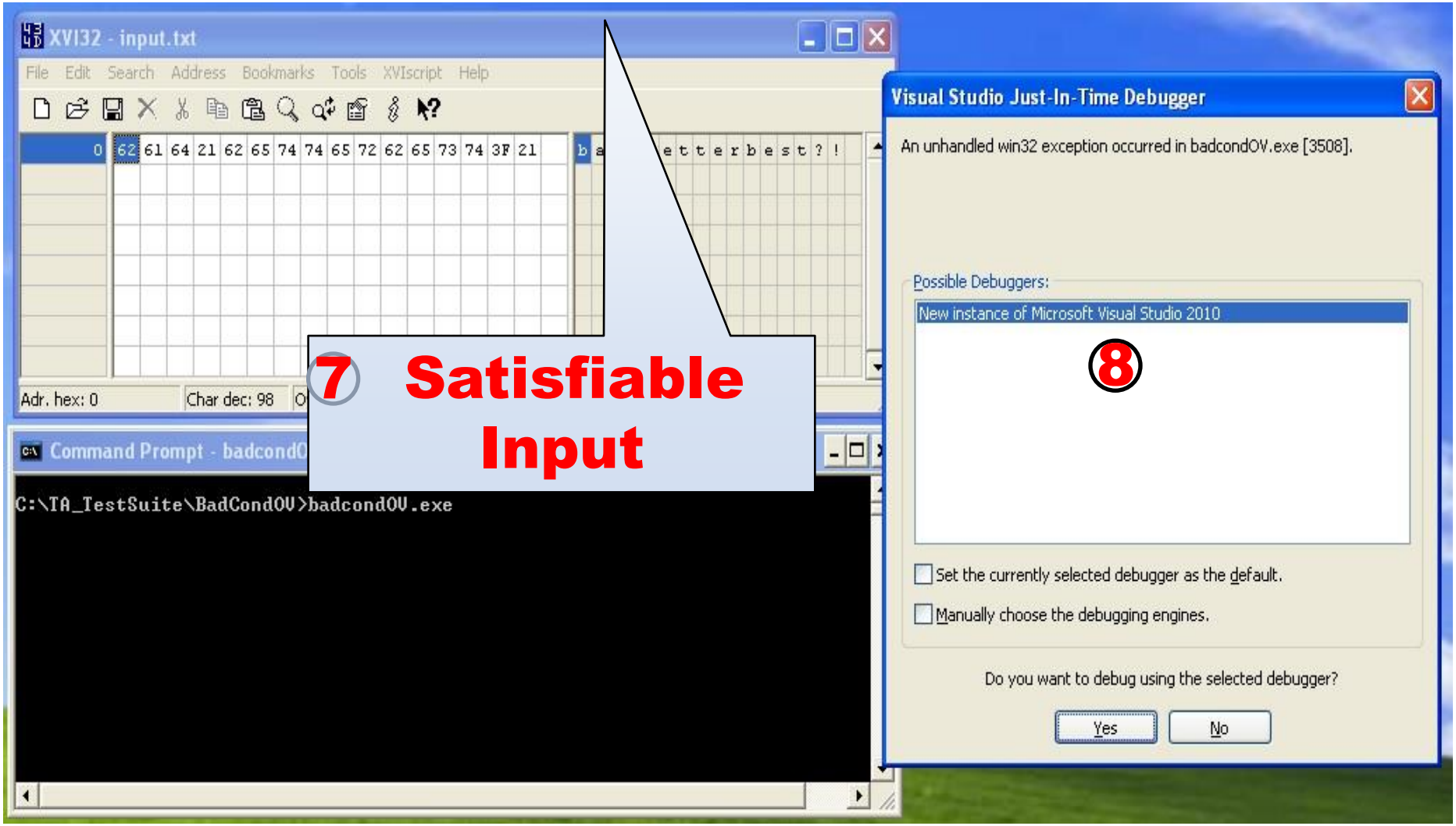
‘a’

‘d’

On-demand Symbolic Execution (What Happens Behind the Scene)



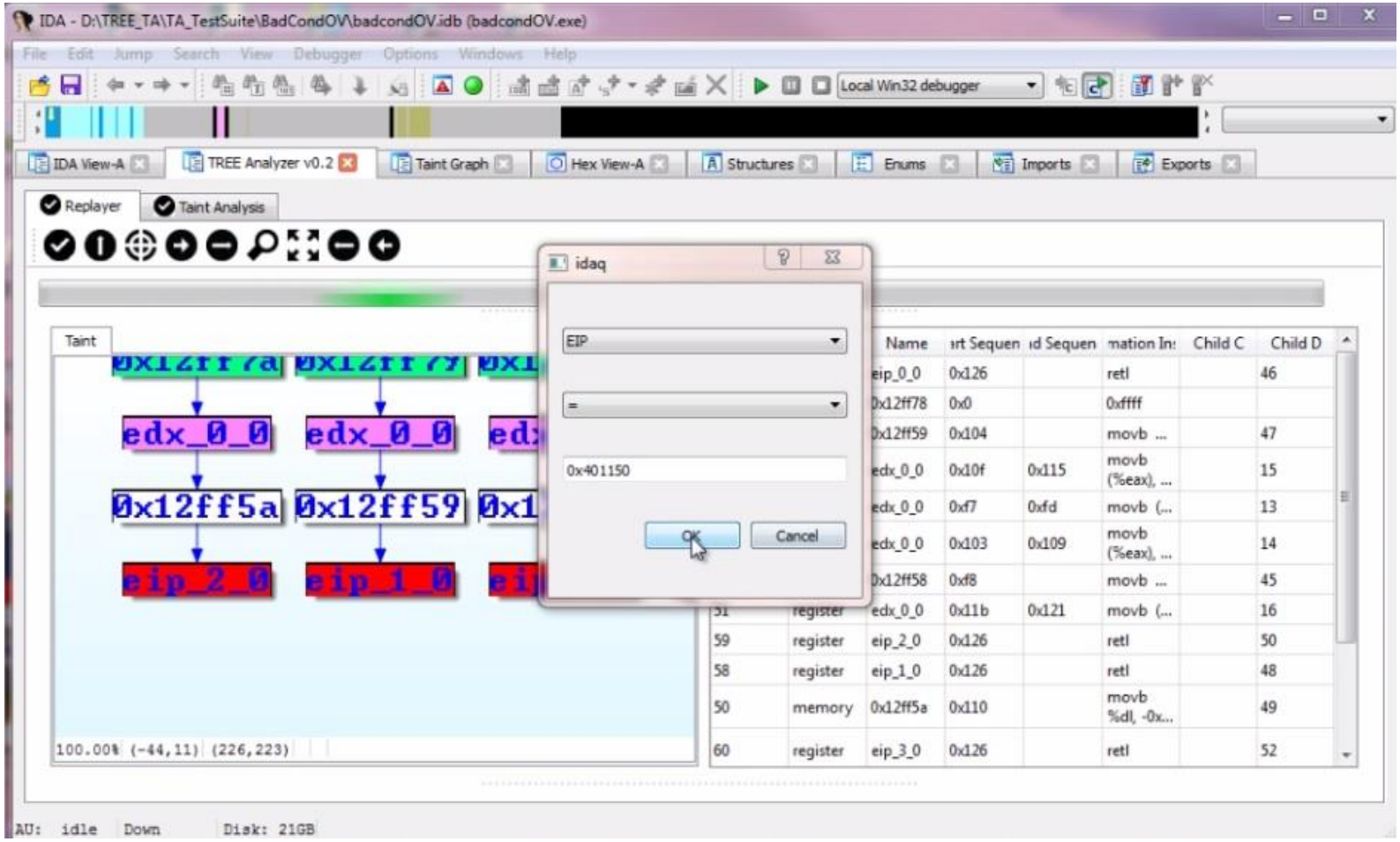
⑧ TREE: Re-execute with “Satisfiable” Input



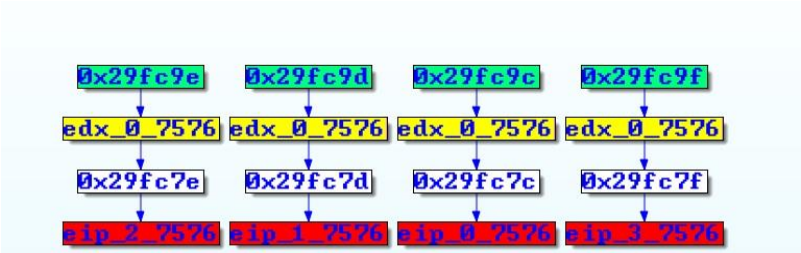
Task 2: Own the Execution
Assume Payload at 0x401150

```
.text:00401144 ; ---------------------------------------------------------------
.text:00401145                 align 10h
.text:00401150                 push    ebp
.text:00401151                 mov     ebp, esp
.text:00401153                 push    1010h
.text:00401158                 push    offset aYouHaveBeenHac ; "*** Yo
.text:0040115D                 push    offset aCbassCrossPlat ; "CBASS(
.text:00401162                 push    0
.text:00401164                 call    ds:MessageBoxA
.text:0040116A                 push    0FFFFFFFFh
.text:0040116C                 call    ds:exit
.text:00401172 ; ---------------------------------------------------------------
```

TREE Constraint Dialogue



Task 2:
Own the Execution: From Crash to Exploit



Symbolize Input and perform concrete-symbolic execution

Symbolic eip =

(= expr_0 (concat (bvand (bvor _IN_0x12ff6c_0xd_SEQ0 (_ bv0 8)) (_ bv255 8)) (bvand (bvor _IN_0x12ff6c_0xc_SEQ0 (_ bv0 8)) (_ bv255 8))))

Query:

get-value (_IN_0x12ff6c_0xd_SEQ0 _IN_0x12ff6c_0xc_SEQ0 _IN_0x12ff6c_0xe_SEQ0 _IN_0x12ff6c_0xf_SEQ0)

SMT Solver

Sat:

(_IN_0x12ff6c_0xd_SEQ0 #x11
_IN_0x12ff6c_0xc_SEQ0 #x50
_IN_0x12ff6c_0xe_SEQ0 #x40
_IN_0x12ff6c_0xf_SEQ0 #x00

TREE/CBASS Demo

Using CBASS/TREE to Explore Bad Paths and Refine Exploits

Real World Case Studies

Target Vulnerability	Vulnerability Name	Target Application Mode	Target OS
CVE-2005-4560	Windows WMF	User Mode	Windows
CVE-2207-0038	ANI Vulnerability	User Mode	Windows
OSVDB-2939	AudioCoder Vulnerability	User Mode	Windows
CVE-2011-1985	Win32k Kernel Null Pointer De-reference	Kernel Mode	Windows
CVE-2004-0557	Sound eXchange (SoX) WAV Multiple Buffer Overflow	User Mode	Linux
Compression/ Decompression	Zip on Android	User Mode	Real Device Trace Generation

Conclusions

- **Dynamic taint analysis and symbolic analysis**
  - Their success in vulnerability discovery
  - Challenges in exploitation analysis
- **Our analysis platform**
  - CBASS (Cross-platform Symbolic-execution System)
    - Supporting automated analysis
  - TREE (Taint-enabled Reverse Engineering Environment)
    - Supporting interactive analysis

Acknowledgements:

- Thanks to Battelle Cyber Innovation Team, particularly:
  - James E. Just, Program Manager
  - Xing Li, Senior Developer
  - Loc Nguyen, Developer

Questions?

Contacts:

li.l.lixin@gmail.com
chaowang@vt.edu