

# **How to Grow a TREE from CBASS**

***Interactive Binary Analysis for  
Security Professionals***

Lixin (Nathan) Li, Xing Li, Loc Nguyen,  
James E. Just

# Outline

- **Background**
- **Interactive Binary Analysis with TREE and CBASS**
- **Demonstrations**
- **Conclusions**

# Interactive Binary Analysis

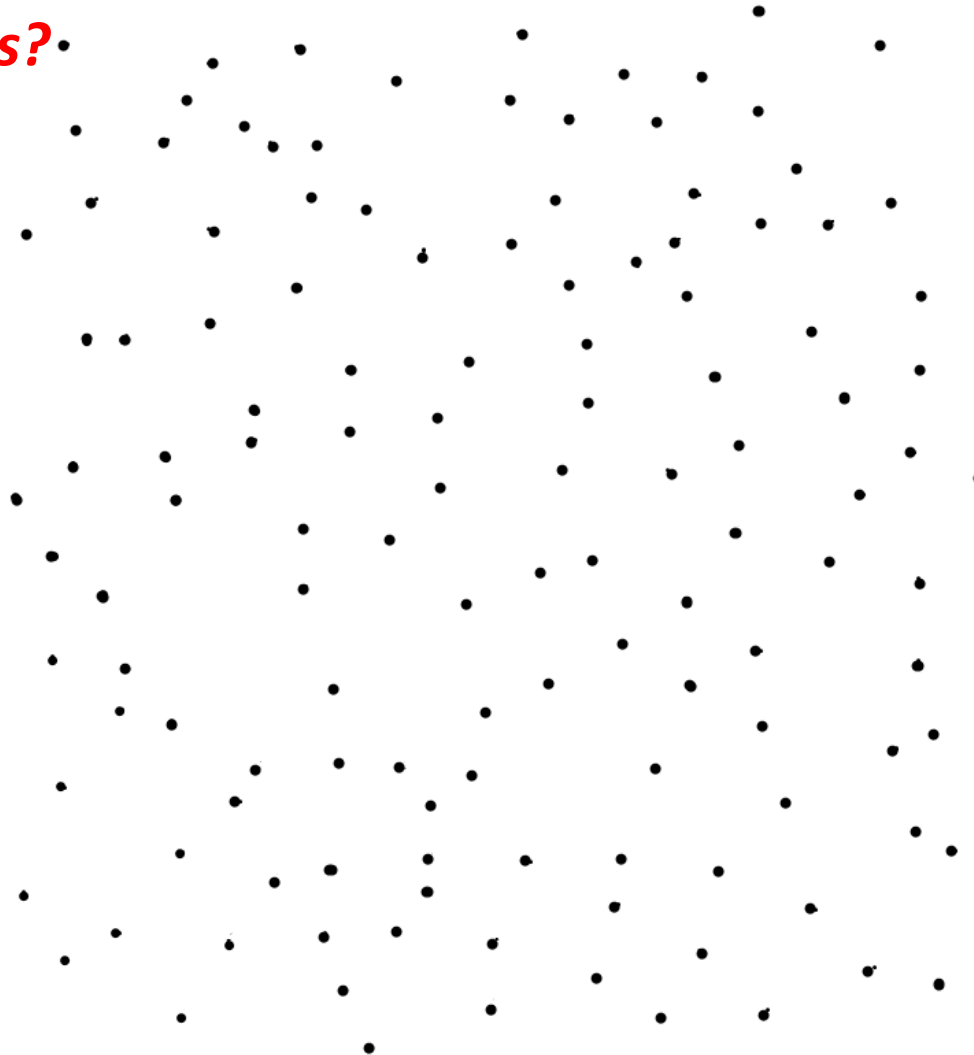
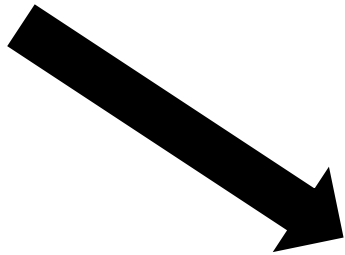
- Automated binary analyses useful for certain tasks (e.g., finding crashes)
- Many binary analyses can't be automated
- Expert experience and heuristics are still **key** to binary analyses

# Benefits of Interactive Binary Analysis

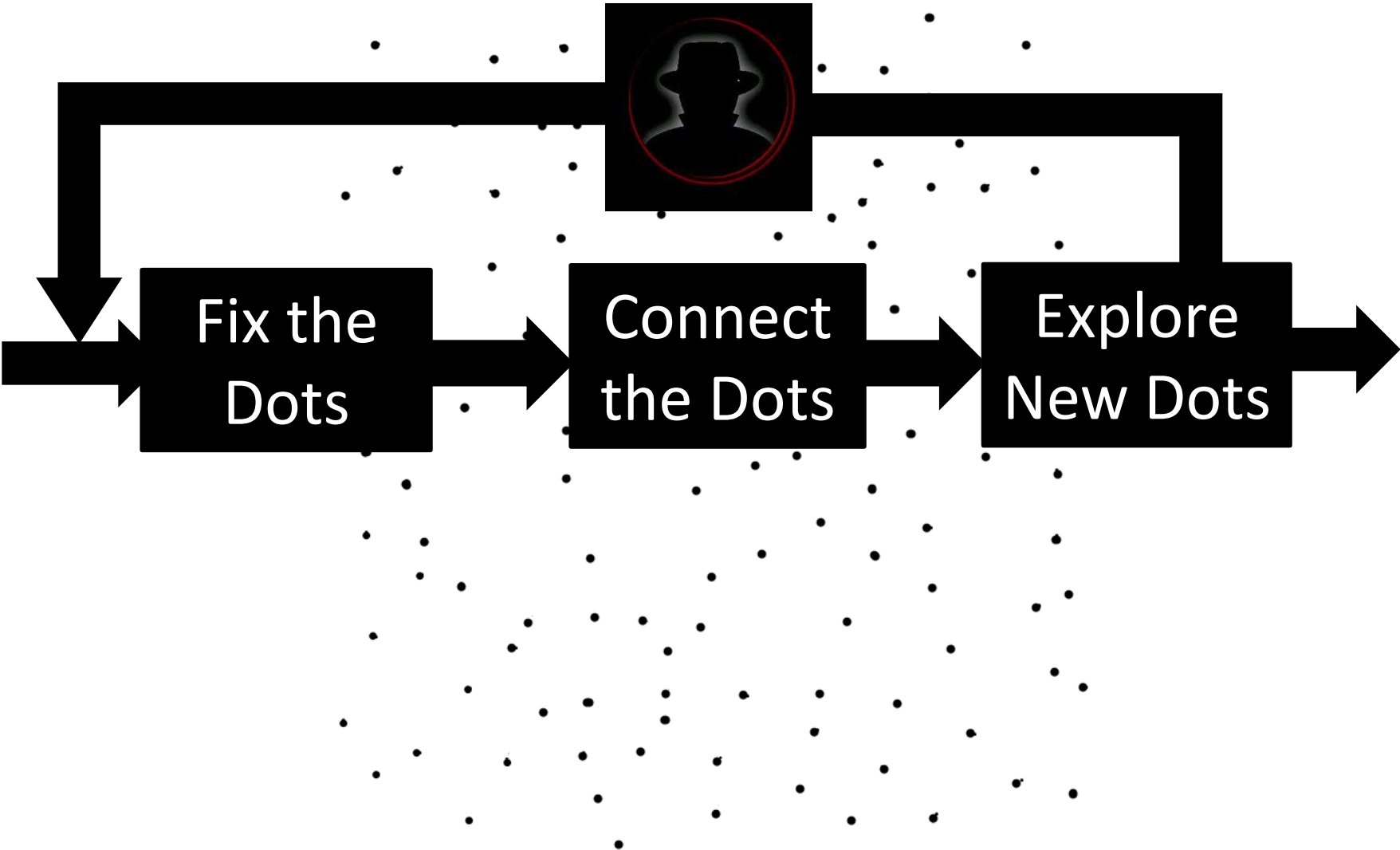
- Applicable to many security problems
- Our tools increase productivity in:
  - Finding vulnerabilities
  - Analyzing root causes
  - Exploitability and risk assessment

# Interactive Analysis Like Connecting Dots

*What's in the dots?*



# Our Tools are Designed to Help



# What Do Our Tools Do?



Fix the  
Dots

Connect  
the Dots

Explore  
New Dots

**TREE**

**Replay & Taint Analysis**

*Tainted-enabled Reverse  
Engineering Environment*

**CBASS**

**Symbolic Execution**

*Cross-platform Binary  
Automated Symbolic  
execution System*

# Gaps between Research and Interactive Binary Analysis

- Existing research does not support interactive binary analysis
  - No practical tools
  - No uniform trace collection tools
  - No unified Instruction Set Architecture (ISA) -independent analysis tools



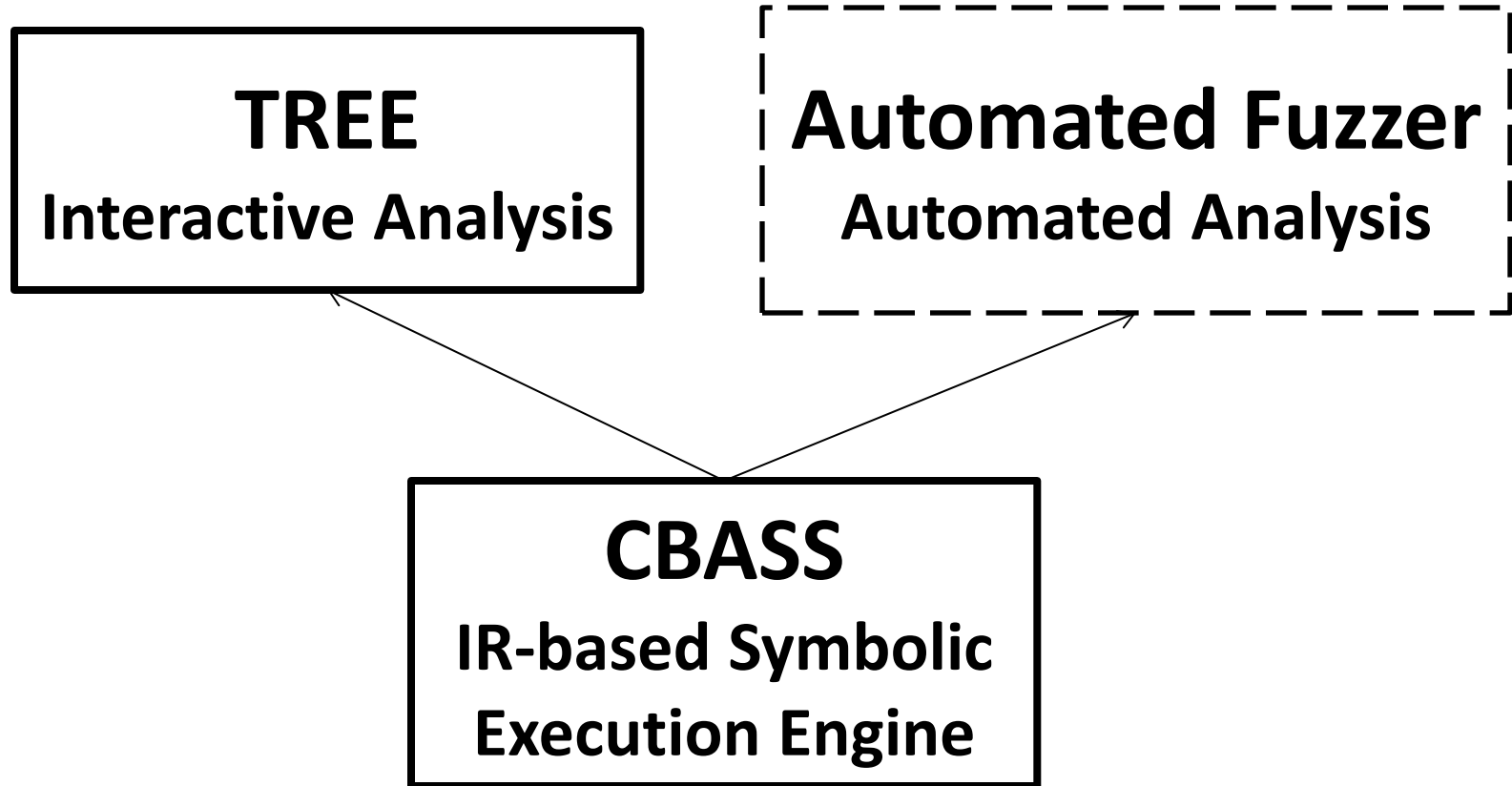
# Bringing Proven Research Techniques to Interactive Binary Analysis

- Our tools use dynamic, trace-based, offline analysis approach
  - Interactive binary analysis [1]
  - Dynamic taint analysis ([2][3][4])
  - Symbolic execution/ SMT solver ([2][5])
  - Trace replay ([6])

# Making It Practical

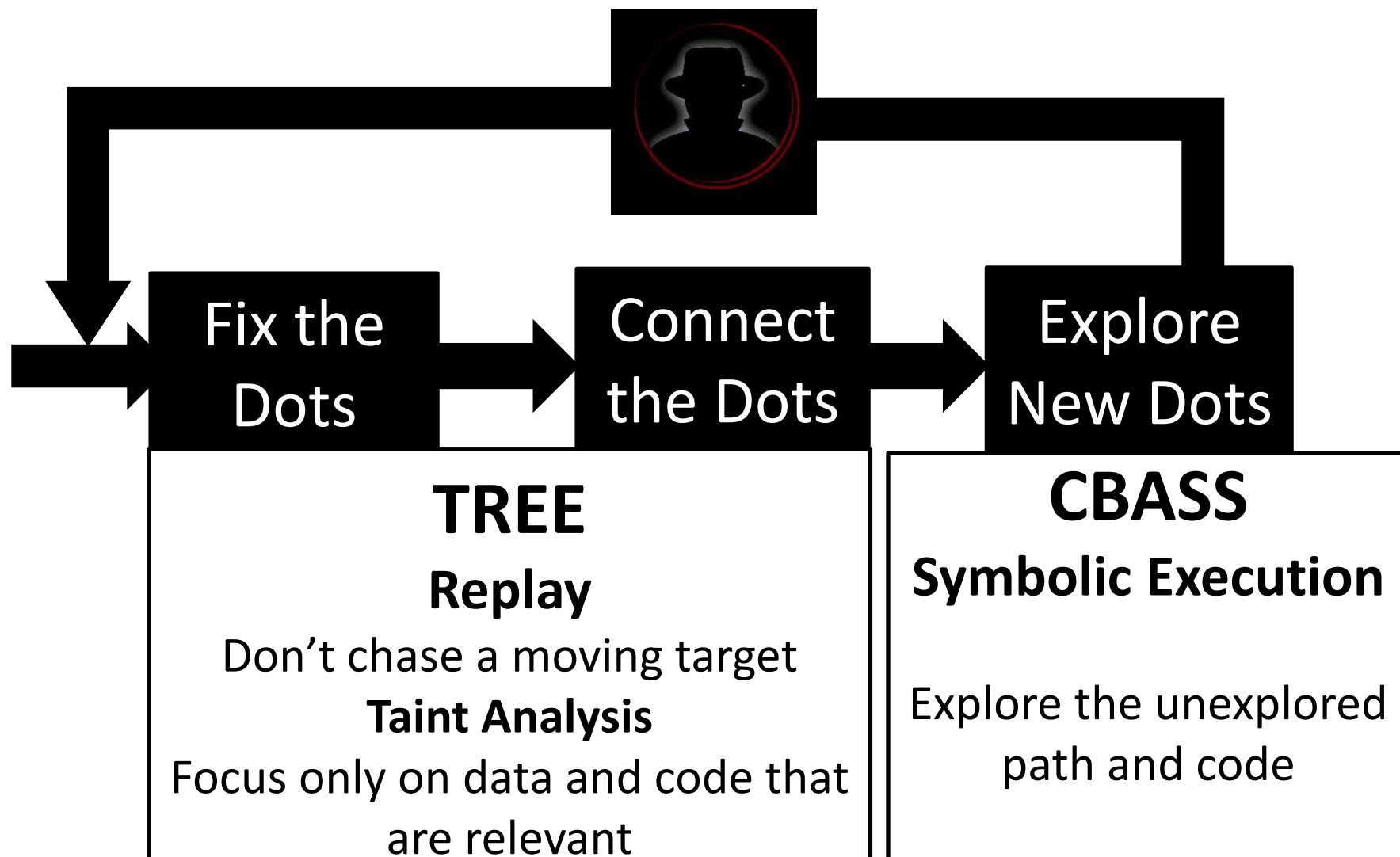
- TREE integrates with IDA Pro now and other mainstream binary analysis environments (later)
- TREE leverages debugging infrastructure to support tracing on multiple platforms
- CBASS uses Intermediate Representation (REIL [6][7])-based approach to support ISA-independent analysis

# CBASS Supports Both Automated & Interactive Analysis



**TREE fills gaps for interactive analysis**

# Tools Support Interactive Binary Analyses



# Illustrative Dots in Vulnerability Analysis: A Running Example

```
//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!");
```

```
//Vulnerable Function
```

```
void StackOVflow(char *sBig,int num)
```

```
{
```

```
    char sBuf[8]= {0};
```

```
    .....
```

```
    for(int i=0;i<num;i++)
```

```
    //Overflow when num>8
```

```
{
```

```
        sBuf[i] = sBig[i];
```

```
}
```

```
.....
```

```
return;
```

```
}
```

# **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
  - ASLR guarantees nothing 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)**

# Connecting Dots is Hard

- Basic elements 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 likely separated by millions of instructions, spatially and temporally
- Multiple protections built in

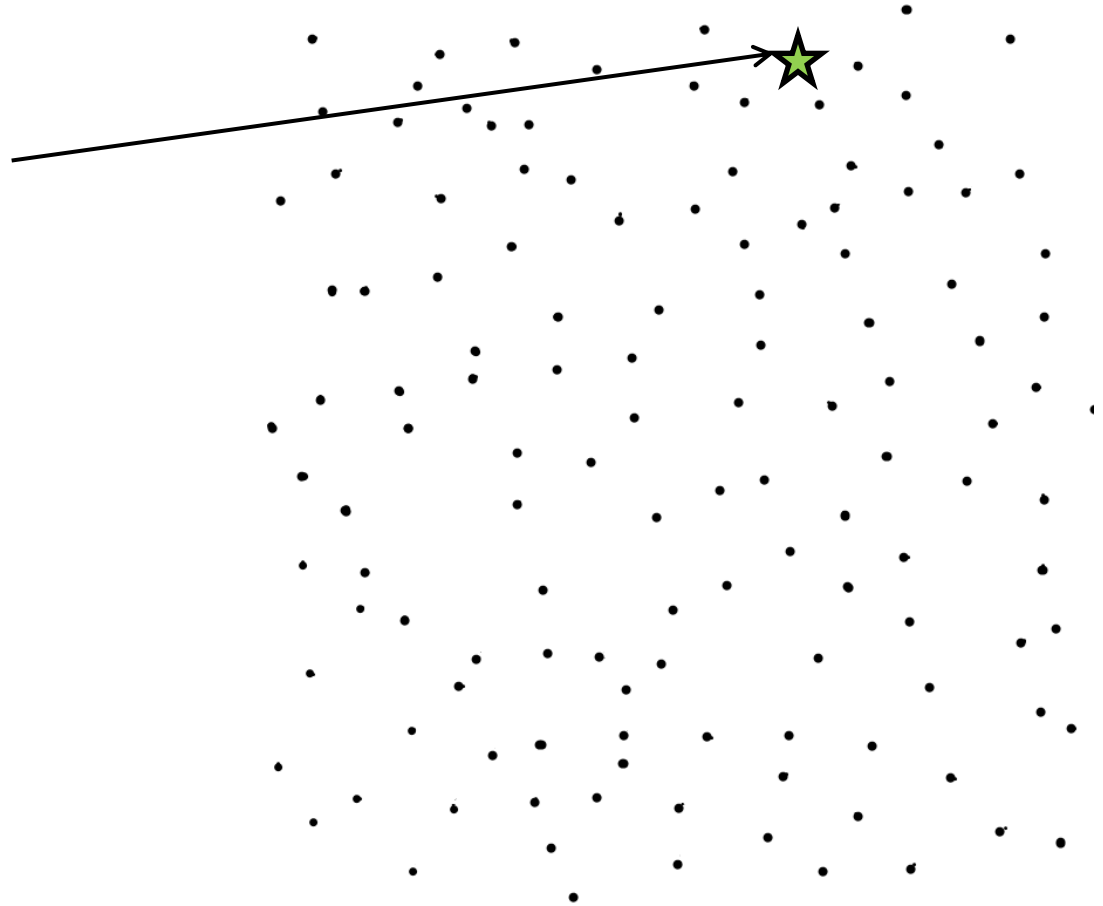
# Techniques Help Connect the Dots

- 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

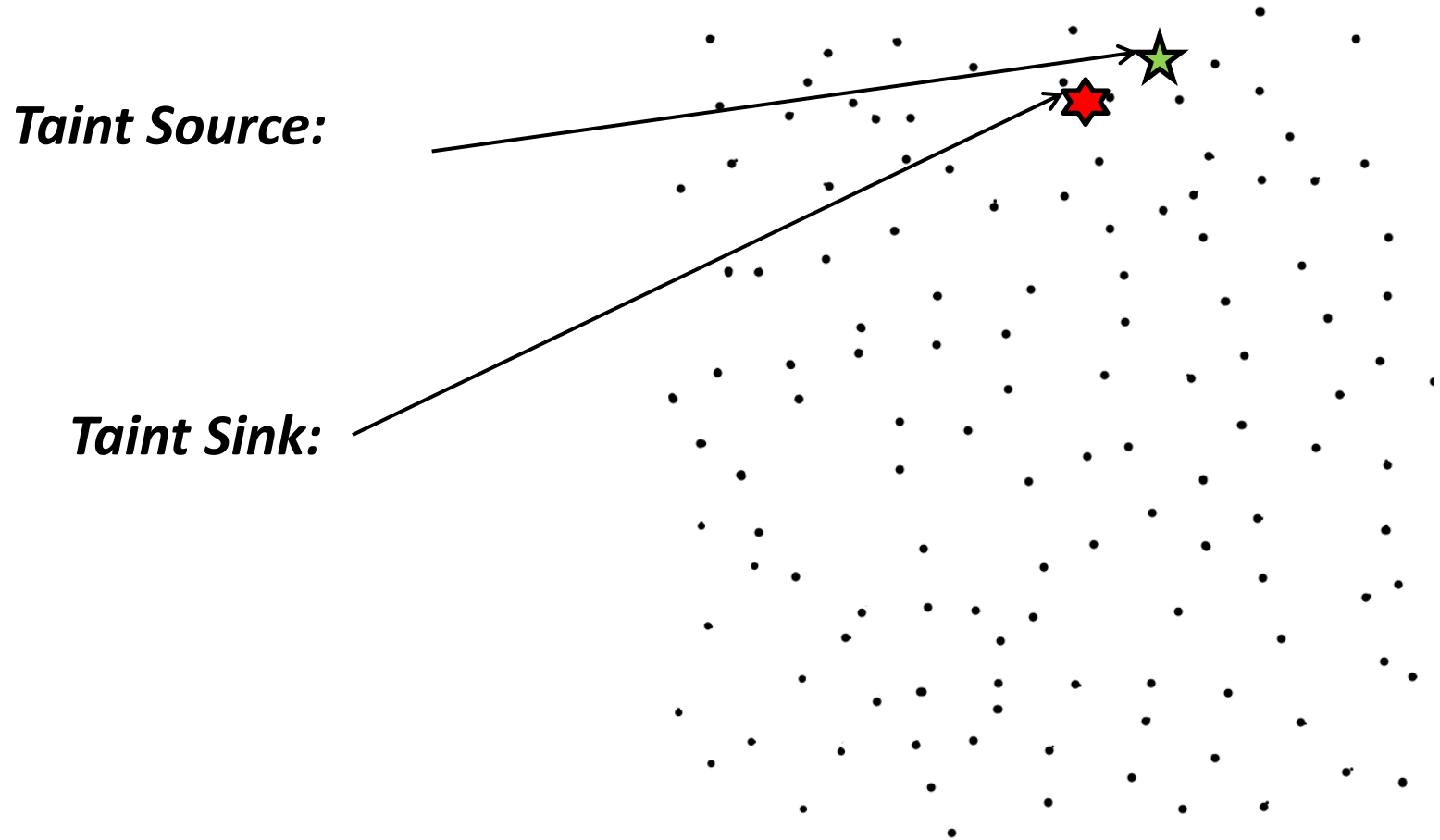
- TREE connects dots -- using taint analysis

*Taint Source:*



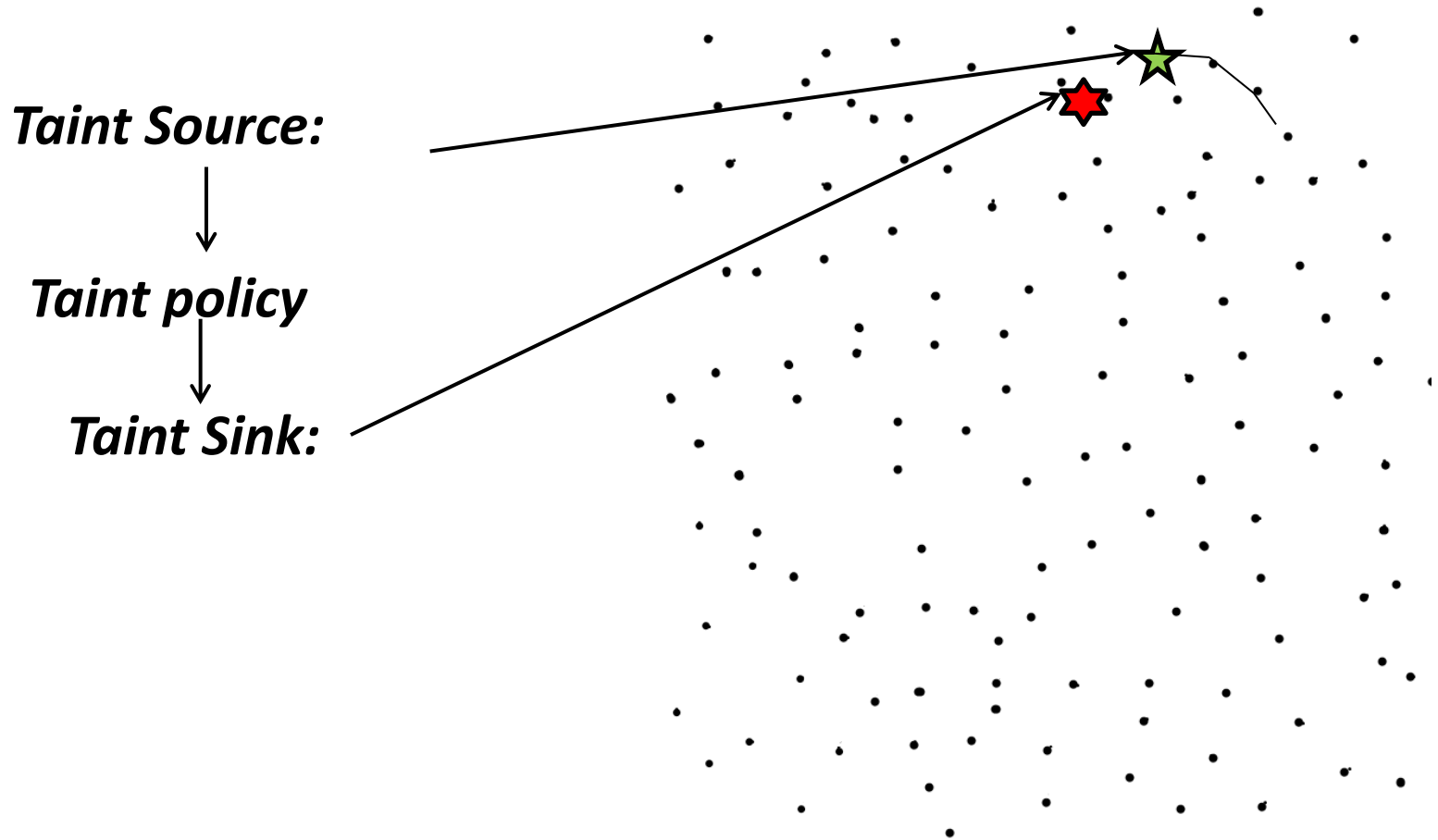
# Connect the Dots

- TREE connects dots -- using taint analysis



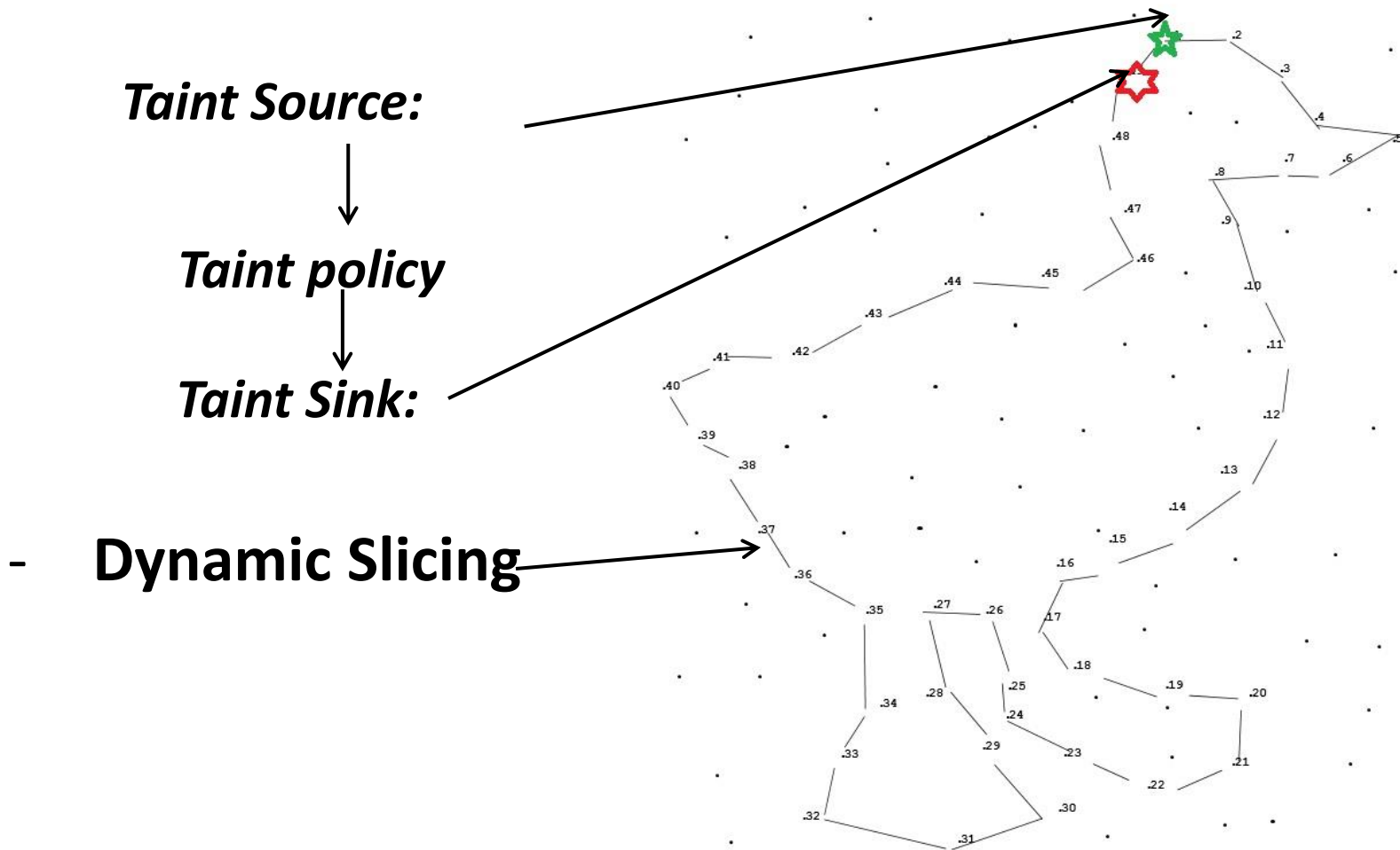
# Connect the Dots

- TREE connects dots -- using taint analysis



# Connect the Dots

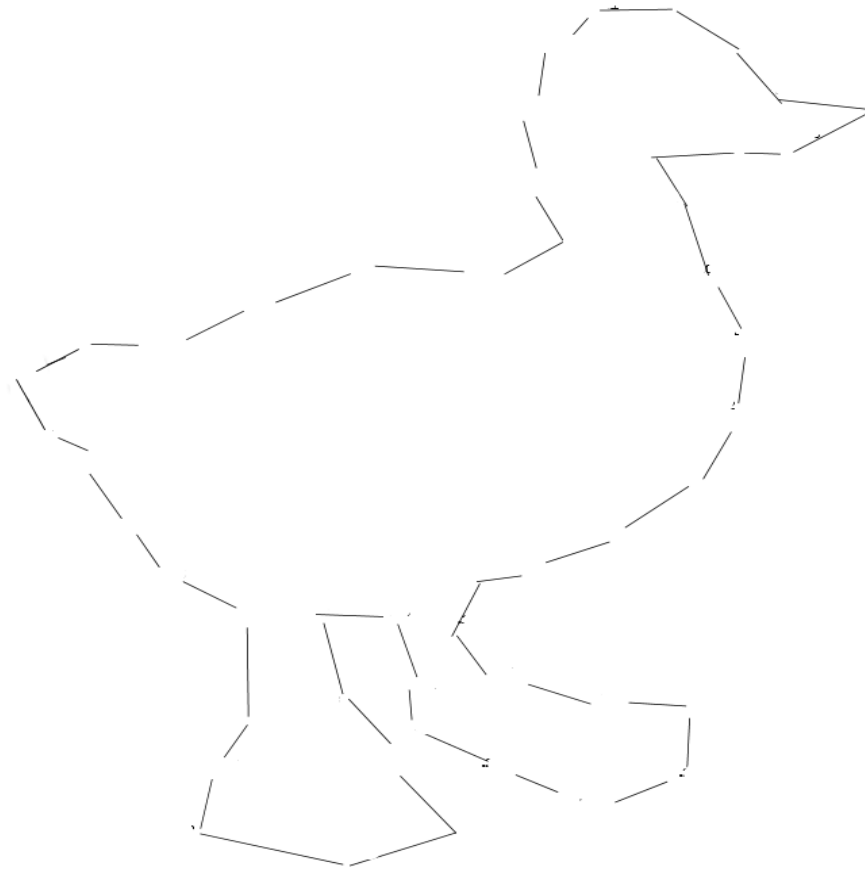
- TREE connects dots -- using taint analysis





# 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)*

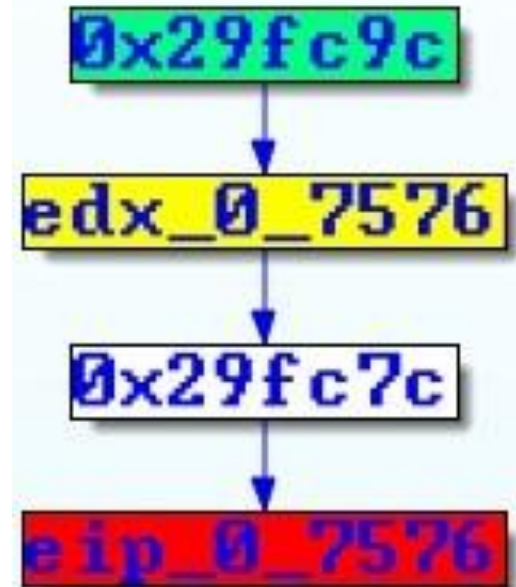


*Taint Sink: eip*

**The Slice**

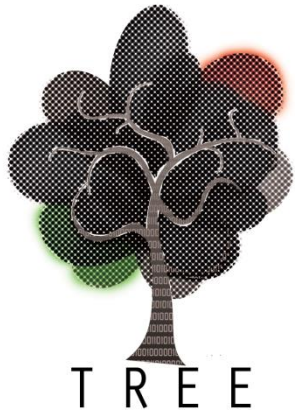
```
call ds:ReadFile
  ↓
movb (%eax), %dl
  ↓
movb %dl,
-0x8(%ebp,%ecx,1)
  ↓
retl
```

**The Taint Graph**



# What You Connect is What You Get

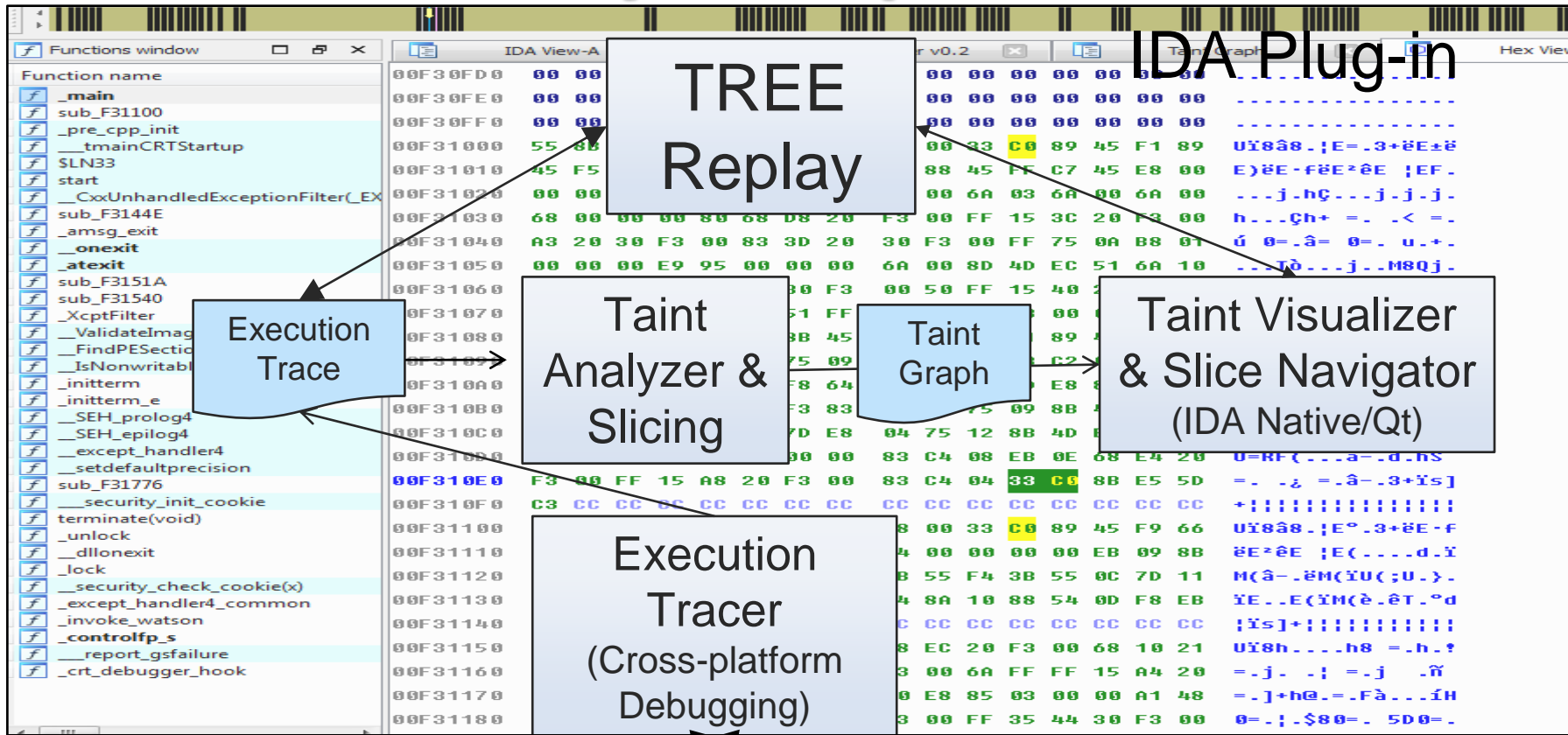
- Dots can be connected in different ways
  - Data dependency
  - Address dependency
  - Branch conditions
  - Loop counter
- Connect dots in different taint policies



**TREE**

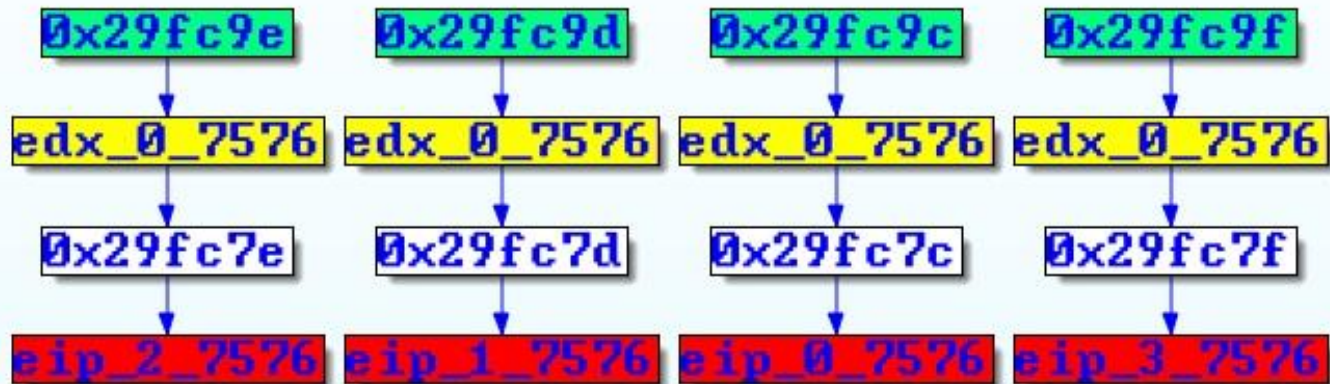
***TAINT-ENABLED  
REVERSE ENGINEERING ENVIRONMENT***

# 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

The screenshot displays the TREE analysis tool interface. At the top, there are three tabs: "Stack View", "Register View", and "Memory View". The "Memory View" is currently selected, showing a list of memory addresses and their corresponding hexadecimal values. A callout box with the text "Register/stack/memory Views" points to the data area. At the bottom, there is a command line with the text "UNKNOWN 0038F798: Stack[00003A1C]:0038F798".

Address	Value
0038F758	0C 3F 7D 77 2C 85 4A 74 80 93 2F 0
0038F768	00 00 00 00 F0 53 7D 77 5C F7 38 0
0038F778	F0 F7 38 00 23 41 87 77 B4 4D 0F 0
0038F788	0C 3F 7D 77 70 10 2F 01 5C 00 00 0
0038F798	10 00 00 00 A8 F7 38 00 00 00 00 0
0038F7A8	10 00 00 00 62 61 64 21 62 65 74 7
0038F7B8	73 74 74 68 00 F8 38 00 E1 12 2F 0
0038F7C8	A0 1B 45 00 38 20 45 00 40 EA 7A 7
0038F7D8	00 00 00 00 00 E0 FD 7E 00 00 00 0
0038F7E8	D0 F7 38 00 B2 3E 7E 4A 3C F8 38 0

UNKNOWN 0038F798: Stack[00003A1C]:0038F798

# TREE: The Front-end of Our Interactive Analysis System

## *Replay is focal point of user interaction*

Replayer Taint Analysis

95%

Taint

85.44% (-2, -91) (721, 322) 3.0

UUID	Type	Name	Start Sequence	End Sequence	Information Instr.	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		

Stack View Register View Memory View

```

0038f758 0c 3f 7d 77 2c 85 4a 74 80 33 2f 0
0038f768 00 00 00 00 f0 53 7d 77 5c f7 38 0
0038f778 f0 f7 38 00 23 41 87 77 b4 4d 0f 0
0038f788 0c 3f 7d 77 70 10 2f 01 5c 00 00 0
0038f798 10 00 00 00 a8 f7 38 00 00 00 00 0
0038f7a8 10 00 00 00 62 61 64 21 62 65 74 7
0038f7b8 73 74 74 68 00 f8 38 00 e1 12 2f 0
0038f7c8 a0 1b 45 00 38 20 45 00 40 ea 7a 7
0038f7d8 00 00 00 00 e0 fd 7e 00 00 00 0 0
0038f7e8 d0 f7 38 00 b2 3e 7e 4a 3c f8 38 0
UNKNOWN 0038f798: Stack[00003a1c]:0038f798

```

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

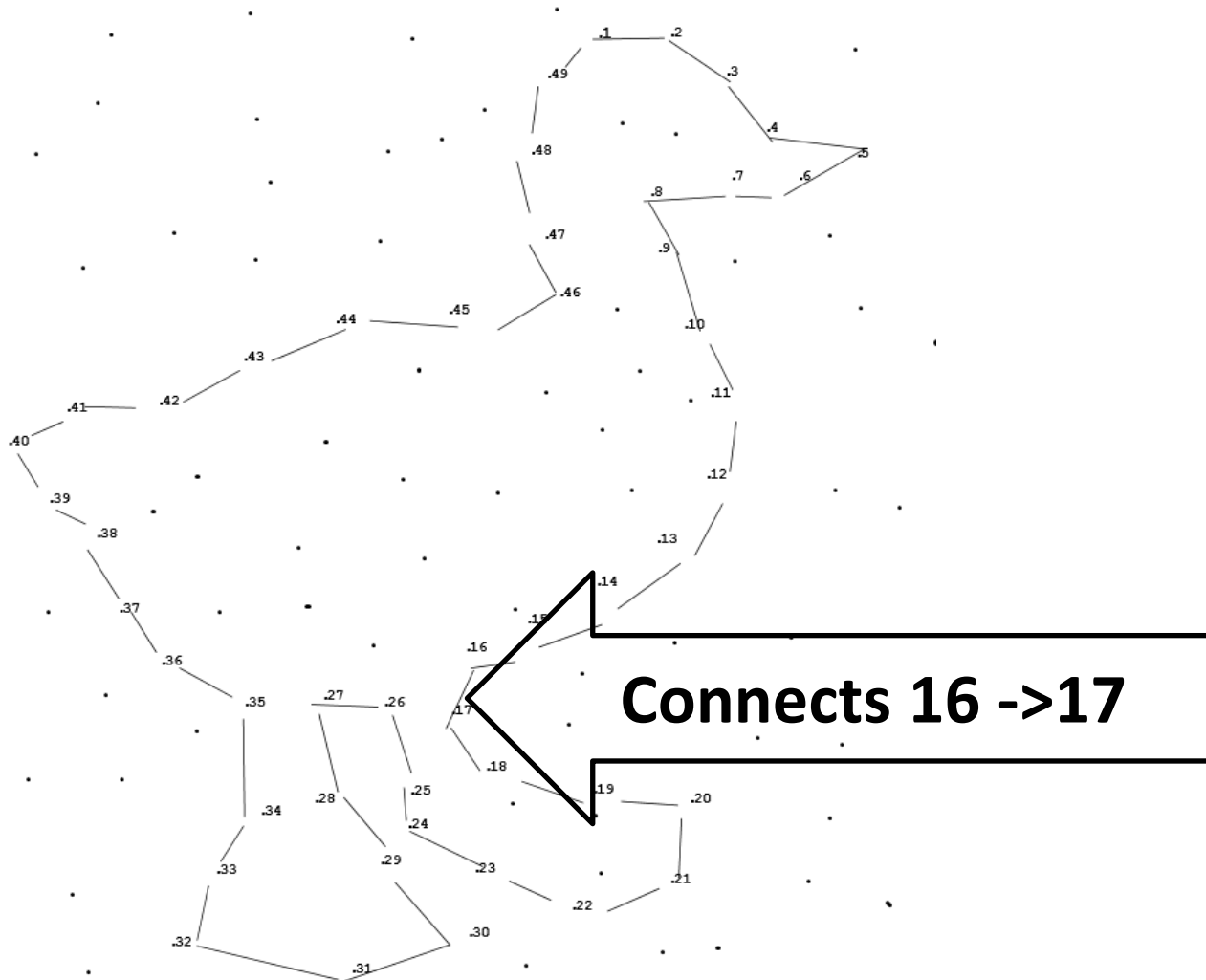
# **Tree Demo**

**Using TREE to Analyze a Crash**

# **Our Tools Support**

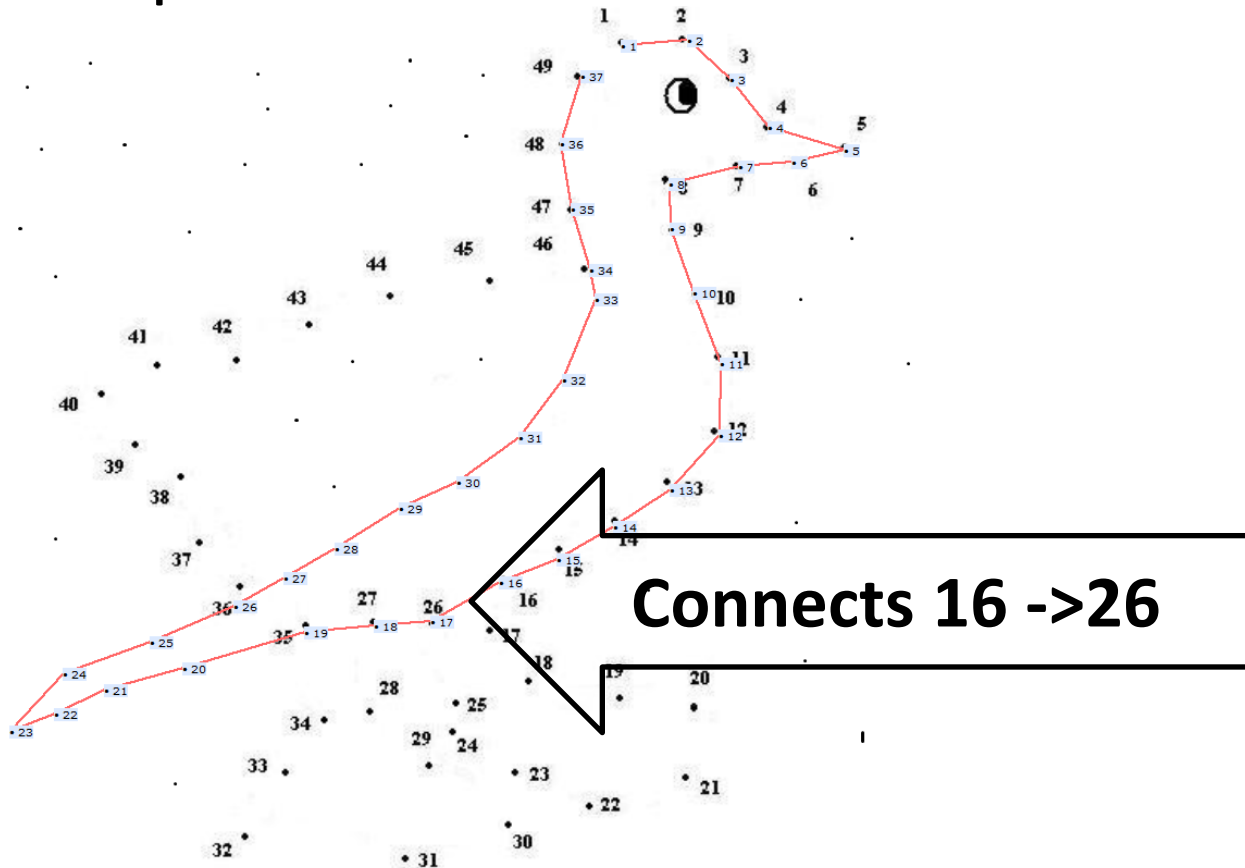
**Exploring New Dots**

# 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?

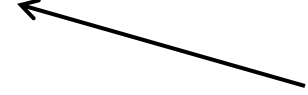
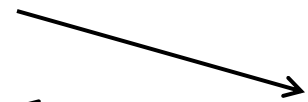
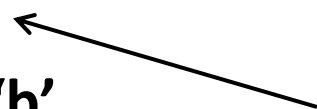
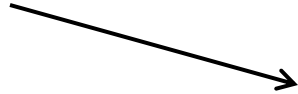
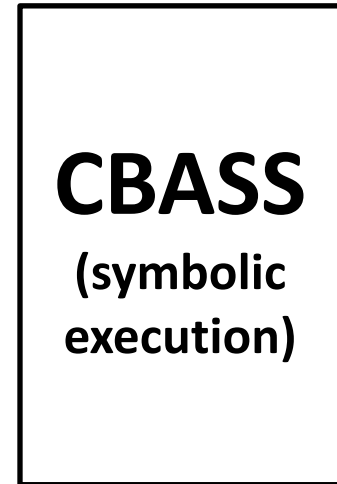
*User:*

*How to execute  
different path  
at branch Y?*

TREE: Input  
[0] must be 'b'

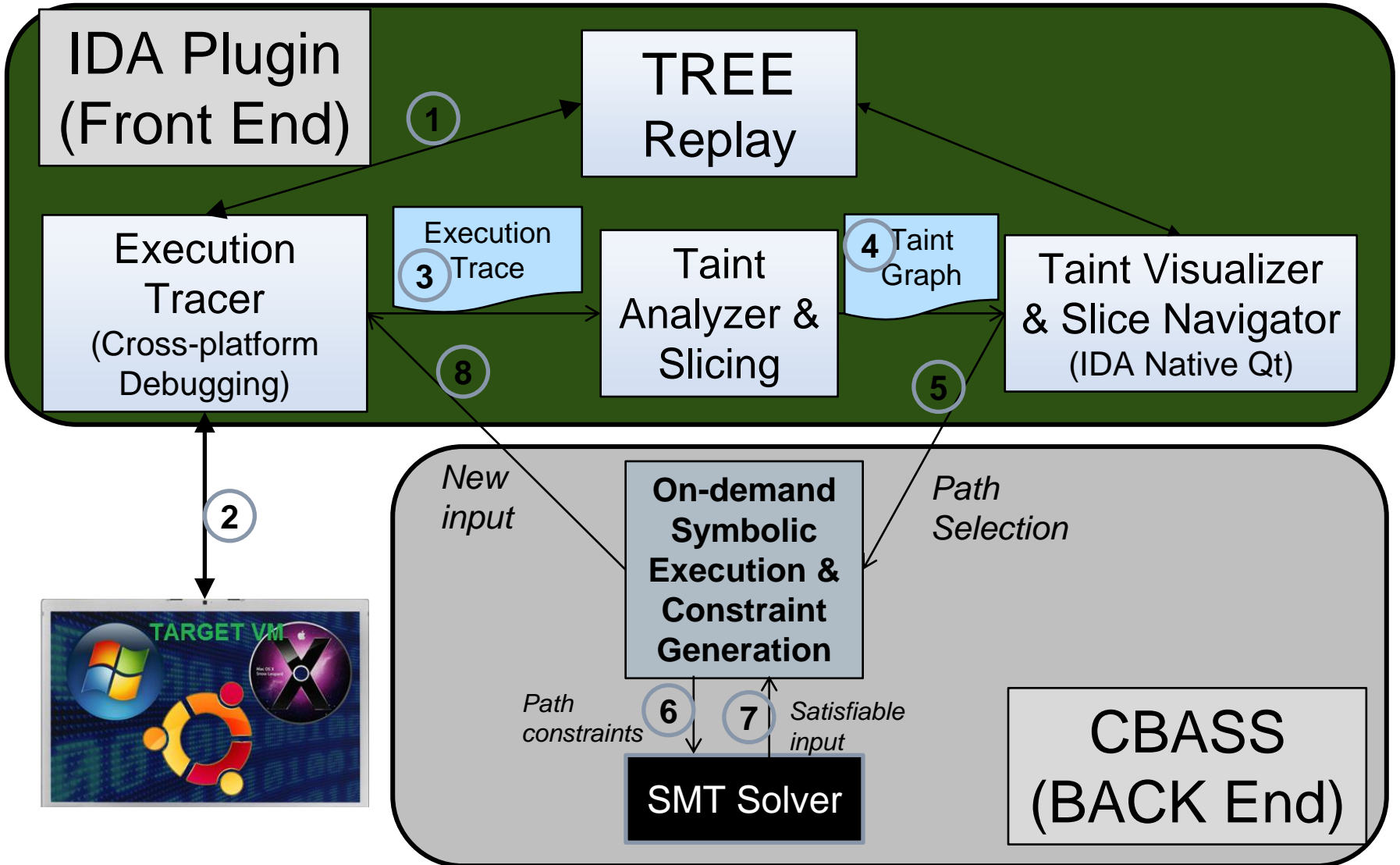
*TREE: Can  
we negate  
path  
condition  
at Y?*

CBASS:  
This byte  
must be 'b'





# 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**

```
StackOverflow(sBigBuf,dwBytesRead)
```

```
else
```

```
printf(“Good!”);
```

## Branch Conditions In Disassembly

```
movsx  edx, [ebp+Buffer]  
cmp    edx, 62h  
jnz    short loc_F3108F
```

```
mov    eax, [ebp+var_18]  
add    eax, 1  
mov    [ebp+var_18], eax
```

```
loc_F3108F:  
movsx  ecx, byte ptr [ebp+var_F]  
cmp    ecx, 61h  
jnz    short loc_F310A1
```

```
mov    edx, [ebp+var_18]  
add    edx, 1  
mov    [ebp+var_18], edx
```

```
loc_F310A1:  
movsx  eax, byte ptr [ebp+var_F+1]  
cmp    eax, 64h  
jnz    short loc_F310B3
```

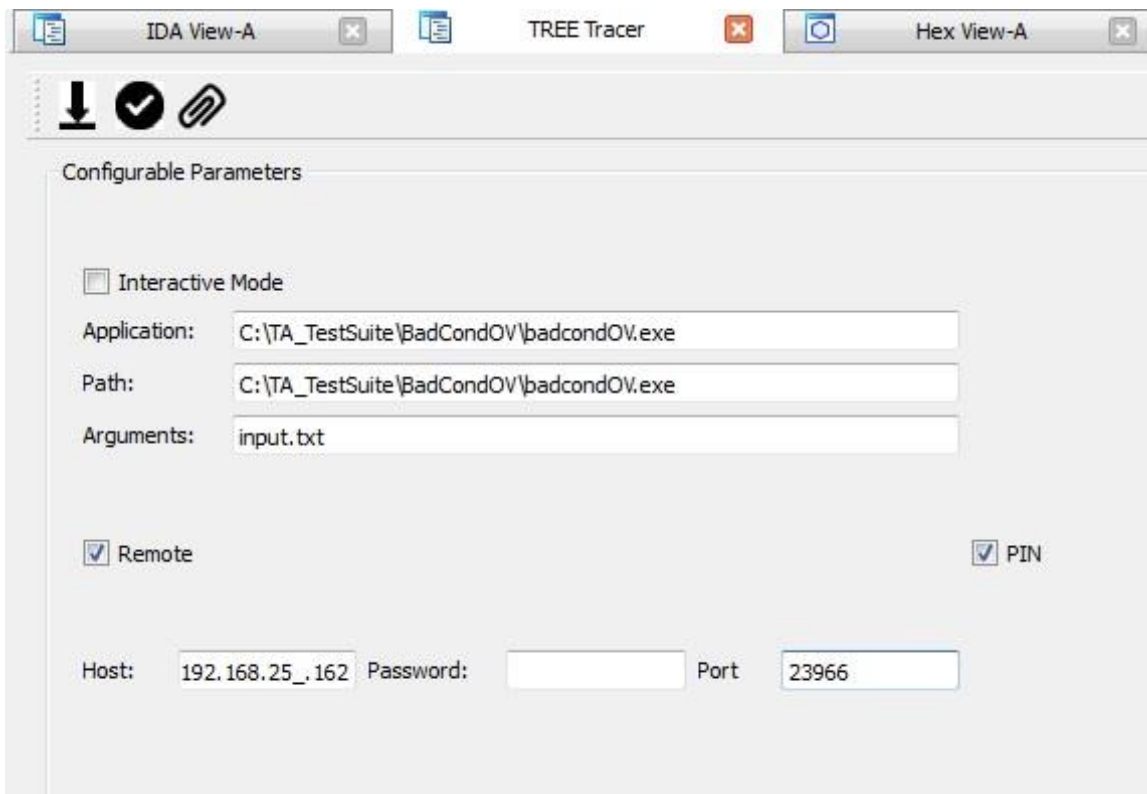
```
mov    ecx, [ebp+var_18]  
add    ecx, 1  
mov    [ebp+var_18], ecx
```

```
loc_F310B3:  
movsx  edx, byte ptr [ebp+var_F+2]  
cmp    edx, 21h  
jnz    short loc_F310C5
```

# 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>



## ② TREE Console: Trace Generation

PINAgent: Connects TREE with PIN tracer

The screenshot displays a Windows Explorer window with the address bar set to `C:\TA_TestSuite\BadCondOV`. The left pane shows a tree view of folders, including `pin_0502`, `pin_xtaint`, `PinAgent`, `PinTree`, `POPPeeper`, `Program Files`, `Python27`, `RE`, `RECON`, `sage`, `simpleov`, `simpleOVCond`, `SoulseekNS`, `Steamcast`, and `TA_TestSuite`. The `TA_TestSuite` folder is expanded, showing subfolders like `BadCondOV`, `BadCondPlusOV`, `BasicOV`, and several `BasicOV_*` folders.

The right pane shows two files: `badcondOV` (C++ Source, 2 KB) and `badcondOV` (executable icon).

In the foreground, a terminal window titled `C:\ Shortcut to PinAgent` shows the following output:

```
PinTrace Remote Agent: nathan-81eb362b [192.168.25.162] on port 23966
Client connected
Worker thread created
Bytes received: 49. Message = C:\TA_TestSuite\BadCondOV\badcondOV.exe input.
Spawn A PINed Process...
PINed Process running...
Good!
PINed Process exit...
```

# ③ TREE: Taint Analysis Configuration

Replayer  Taint Analysis

Taint Propagation Policy

- TAINT\_DATA
- TAINT\_BRANCH
- TAINT\_COUNTER
- TAINT\_ADDRESS

Instruction Set Architecture

- x86
- x86\_64
- ARM
- PPC
- MIPS

Misc

- PIN
- Verbose

Image Load Table

Name	Address	Size
['badcon...	0x12f0000	0x5000
ntdll.dll	0x77e00000	0x180000
kernel32.dll	0x777c0000	0x110000
kernel32.dll	0x777c0000	0x110000
KernelBas...	0x75ff0000	0x47000
user32.dll	0x75a70000	0x100000

Taint Source Table

Input Address	Size	
0x38f7ac	16	626164210

Taint Graph Output

```
Path Conditions:  
[19]bc_0x3a[0x3a:0x0] <-jnz 0x9  
  
[18]reg_eflags_0[0x39:0x0]  
[0x3c:0x0] <-cmp $0x62, %edx  
  
[17]reg_edx_0_0[0x38:0x0]  
[0x41:0x0] <-movsxb -0x10(%ebp),  
%edx  
  
[1]in_0x12ff6c[0x0:0x0]  
[0xa5b:0x0] <-0xffff:ReadFile  
[22]bc_0x3d[0x3d:0x0] <-jnz 0x9
```

# 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

# ⑤ Negate 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 ← [b]
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 ← [a]
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 ← [d]
```

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

```
C:\windows\system32\cmd.exe
C:\CBass\src>CBass_basicov_deno_path.bat
```

6

```
(set-logic QF_AUFBV)
```

```
(declare-fun _IN_0x12ff6c_0x0_SEQ0 () (_ BitVec 8))
```

```
(declare-fun EXPR_0 () (_ BitVec 32))
```

```
(assert (= EXPR_0 (bvsub ((_ sign_extend 24) (bvxor _IN_0x12ff6c_0x0_SEQ0 (_ bv128 8)))) (_ bv4294967168 32))))
```

```
(assert (= (ite (not (= (ite (not (= (bvand ((_ extract 63 0) (bvsub ((_ sign_extend 32) (bvand ((_ extract 31 0) EXPR_0) (_ bv4294967295 32)))) (_ bv98 64)))) (_ bv4294967295 64)) (_ bv0 64)))) (_ bv1 32) (_ bv0 32)) (_ bv0 32)) (_ bv1 8) (_ bv0 8)) (_ bv0 8))))
```

```
(check-sat)
```

```
(get-value (_IN_0x12ff6c_0x0_SEQ0))
```

```
RD t5, EMPTY , DWORD edx]]
58:4B1077 cnp edx, 98
[4B107708: and [DWORD edx, DWORD 2147483648, DUORD t0], 4B107781: and [D
WORD 98, DWORD 2147483648, DUORD t1], 4B107782: sub [DUORD edx, DWORD 98, QWORD
t2], 4B107783: and [QWORD t2, QWORD 2147483648, DUORD t3], 4B107784: bsh [DWORD
t3, DWORD -31, BYTE SF], 4B107785: xor [DUORD t0, DUORD t1, DUORD t4], 4B107786:
xor [DUORD t0, DUORD t3, DUORD t5], 4B107787: and [DUORD t4, DUORD t5, DUORD t6
], 4B107788: bsh [DUORD t6, DUORD -31, BYTE OF], 4B107789: and [QWORD t2, QWORD
4294967296, QWORD t7], 4B10778A: bsh [QWORD t7, QWORD -32, BYTE CF], 4B10778B: a
nd [QWORD t2, QWORD 4294967295, DUORD t8], 4B10778C: biaz [DUORD t8, EMPTY , BYT
E ZF]]
59:4B107a jnz loc_4B108E
[4B107A08: biaz [BYTE ZF, EMPTY , BYTE ZF], 4B107A09: jcc [BYTE t0, EMPI
Y , DWORD 41985421]]
Invoke Z3 solver z3.exe /smt2 /m path_conditions.smt2
sat
(<<_IN_0x12ff6c_0x0_SEQ0 #x62))
```

7 Satisfiable Input (0x62, 'b')



# 8 TREE: Re-execute with “Satisfiable” Input

The image shows a screenshot of a debugger interface. In the background, there is a hex editor window titled 'XVI32 - input.txt' showing memory addresses and their corresponding hexadecimal and decimal values. Below it is a Command Prompt window titled 'Command Prompt - badcond0' showing the command 'C:\TA\_TestSuite\BadCond0V>badcond0V.exe'. In the foreground, a 'Visual Studio Just-In-Time Debugger' dialog box is open, displaying an error message: 'An unhandled win32 exception occurred in badcond0V.exe [3508]'. Below the message, a list of 'Possible Debuggers' is shown, with 'New instance of Microsoft Visual Studio 2010' selected. A red circle with the number '8' is overlaid on this list. At the bottom of the dialog, there are checkboxes for 'Set the currently selected debugger as the default' and 'Manually choose the debugging engines', and a question 'Do you want to debug using the selected debugger?' with 'Yes' and 'No' buttons.

**7 Satisfiable Input**

**8**

# 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 ; "CBASSC  
.text:00401162 push 0  
.text:00401164 call ds:MessageBoxA  
.text:0040116A push 0FFFFFFFFh  
.text:0040116C call ds:exit  
.text:00401172 : -----
```

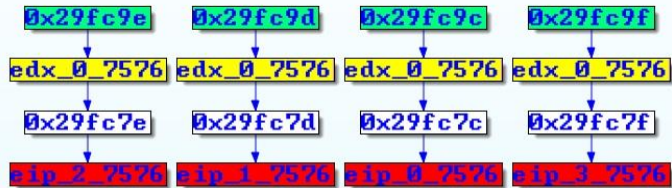
# TREE Constraint Dialogue

The screenshot shows the IDA Pro interface with the TREE Analyzer v0.2 window active. A taint analysis tree is visible, showing nodes for registers like `edx_0_0` and `eip_2_0`. A dialog box titled "idaq" is open, allowing the user to set a constraint for the EIP register. The constraint is set to `= 0x401150`. The dialog box has "OK" and "Cancel" buttons.

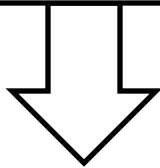
Name	irt Sequen	id Sequen	ation In:	Child C	Child D		
eip_0_0	0x126		retl		46		
0x12ff78	0x0		0xffff				
0x12ff59	0x104		movb ...		47		
edx_0_0	0x10f	0x115	movb (%eax), ...		15		
edx_0_0	0xf7	0xfd	movb (...		13		
edx_0_0	0x103	0x109	movb (%eax), ...		14		
0x12ff58	0xf8		movb ...		45		
edx_0_0	0x11b	0x121	movb (...		16		
59	register	eip_2_0	0x126		retl		50
58	register	eip_1_0	0x126		retl		48
50	memory	0x12ff5a	0x110		movb %dI, -0x...		49
60	register	eip_3_0	0x126		retl		52

# Task 2:

## Own the Execution: From Crash to Exploit



Symbolize Input and perform  
concrete-symbolic execution

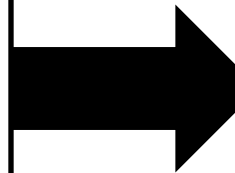


**Symbolic eip =**

```
(= 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

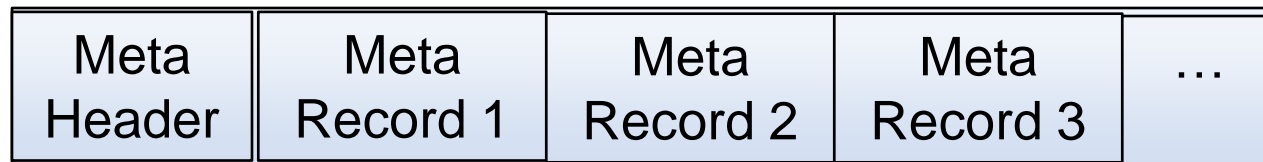
Target Vulnerability	Vulnerability Name	Target Application Mode	Target OS
CVE-2005-4560	Windows WMF	User Mode	Windows
CVE-2207-0038	ANI Vulnerability	User Mode	Windows
OSVDB-2939	AudioCoder Vulnerability	User Mode	Windows
CVE-2011-1985	Win32k Kernel Null Pointer De-reference	Kernel Mode	Windows
CVE-2004-0557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

# Highlights from Real World Case Study: Windows WMF Vulnerability (CVE-2005-4560)

- WMF SETABORTPROC Escape Vulnerability
  - <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-4560>
  - The Windows Graphical Device Interface library (GDI32.DLL) in Microsoft Windows allows remote attackers to execute arbitrary code via a Windows Metafile (WMF) format image with a crafted SETABORTPROC GDI Escape function call, related to the Windows Picture and Fax Viewer (SHIMGVW.DLL).

# WMF Format

- [MS-WMF]: Windows Metafile Format
  - <http://msdn.microsoft.com/en-us/library/cc250370.aspx>
- A Simplified One:
  - <http://wware.sourceforge.net/caolan/ora-wmf.html>
- Overall WMF File Structure:



- One type of record is “escape” record
- SETABORTPROC escape allow an application to register a hook function to handle spooler errors



# WMF Crash

escape - Windows Picture and Fax Viewer

## The WMF SETABORTPROC Vulnerability

```
rundll32.exe c:\windows\system32\shimgvw.dll,ImageView_Fullscreen  
C:\escape\escape.wmf
```

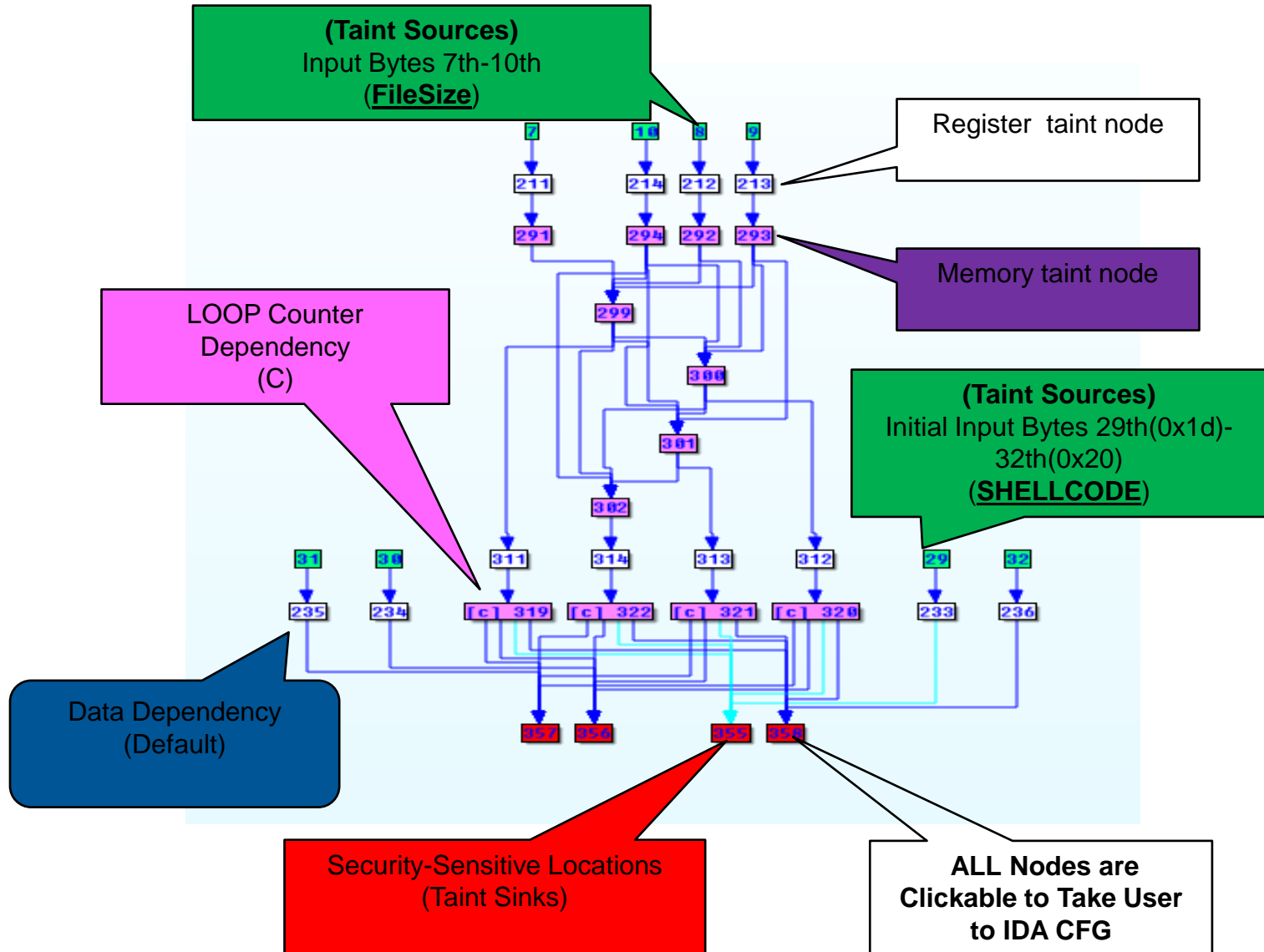
**Dynamic Facts:**  
**229,679**  
instructions  
executed just to  
cause the crash

**Run a DLL as an App**  
Run a DLL as an App has encountered a problem and needs to close. We are sorry for the inconvenience.  
If you were in the middle of something, the information you were working on might be lost.  
**Please tell Microsoft about this problem.**  
We have created an error report that you can send to help us improve Run a DLL as an App. We will treat this report as confidential and anonymous.  
To see what data this error report contains, [click here](#).  
Send Error Report Don't Send

Taskbar icons: Back, Forward, Home, Stop, Refresh, Print, Copy, Paste, Find, Find Next, Close, Help, Run a DLL as an App

# WMF Taint Graph

## Partial TREE Taint Graph Visualization

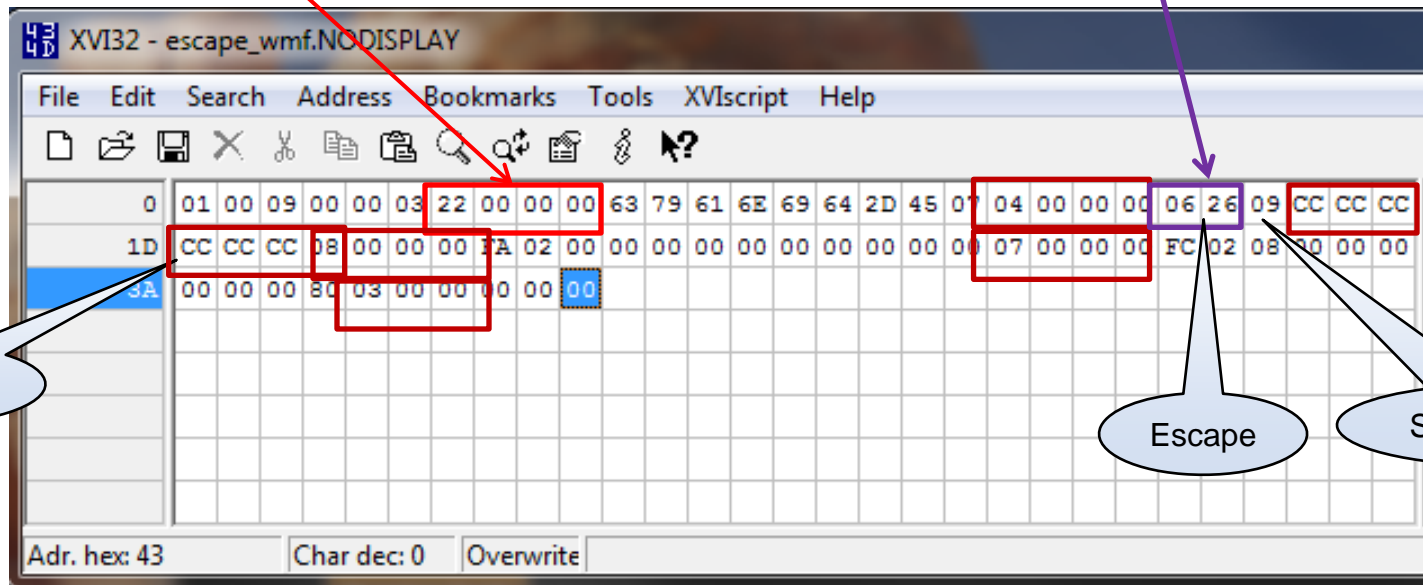


# WMF File: The Fields & The Vulnerability

- Key Structures:

```
typedef struct _WindowsMetaHeader
{
    WORD FileType; /* Type of metafile (0=memory, 1=disk) */
    WORD HeaderSize; /* Size of header in WORDS (always 9) */
    WORD Version; /* Version of Microsoft Windows used */
    DWORD FileSize; /* Total size of the metafile in WORDs */
    WORD NumOfObjects; /* Number of objects in the file */
    DWORD MaxRecordSize; /* The size of largest record in WORDs */
    WORD NumOfParams; /* Not Used (always 0) */
} WMFHEAD;
```

```
typedef struct _StandardMetaRecord
{
    DWORD Size;
    /* Total size of the record in WORDs */
    WORD Function;
    /* Function number (defined in WINDOWS.H) */
    /*
    WORD Parameters[];
    /* Parameter values passed to function */
} WMFRECORD;
```



Shellcode

Escape

SetAbortProc

# WMF Slicing (1)

## An Instruction Slice Traced Back from Crash Site to Input

Each node uniquely trace back to one execution event through its sequence number

0x77f330a3 call eax 2 ffd0 0x0 0x3812f Reg( EAX=0xa8b94 ESP=0xb4fb88 EIP=0x77f330a3 ) W 4 b4fb88

0x77c472e3 rep movsd 2 f3a5 0x0 0xb142 Reg( EDI=0xa8804 eflags=0x10216 ESI=0xa9f8c ECX=0xa9f8c cc\_cc\_cc\_cc W 4 a8804

0x77f2e997 mov ecx, [ebp+arg\_8] 3 8b4d10 0x0 0xc5c3 Reg( EBP=0xb4fbf8 ECX=0x7c809a20 ) R 4 b4fc08 44\_0\_0\_0

0x77f2e983 mov [ebp+arg\_8], eax 3 894510 0x0 0xbd8c Reg( EAX=0x44 EBP=0xb4fbf8 ) W 4 b4fc08

0x77f2e97f add eax, eax 2 03c0 0x0 0xbd89 Reg( EAX=0x22 eflags=0x246 )

0x77f2e949 mov eax, [edi+6] 3 8b4706 0x0 0xbd7d Reg( EAX=0xa8920 EDI=0xa87e8 ) R 4 a87ee22\_0\_0\_0

0x77c472e3 rep movsd 2 f3a5 0x0 0xb13c Reg( EDI=0xa87ec eflags=0x10216 ESI=0xa9f74 ECX=0x10 ) R 4 a9f74 0\_3\_22\_0 W 4 a87ec

# WMF Slicing (2)

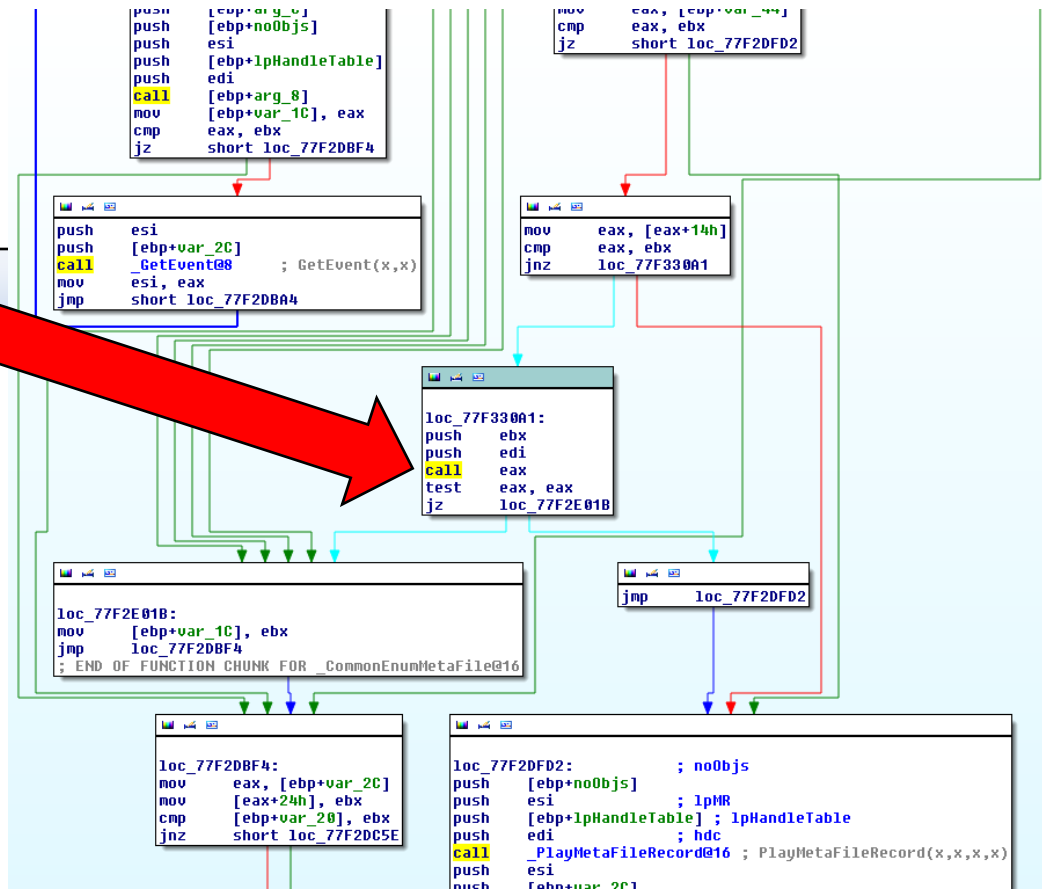
An Instruction Slice with Text Helps

Put Instruction In Its Context Helps  
More

Module: **gdi32.dll**

Function: **CommonEnumMetaFile**

text:77F330A3 call eax



# WMF Slicing (3)

An Instruction Slice with Text Helps a Little

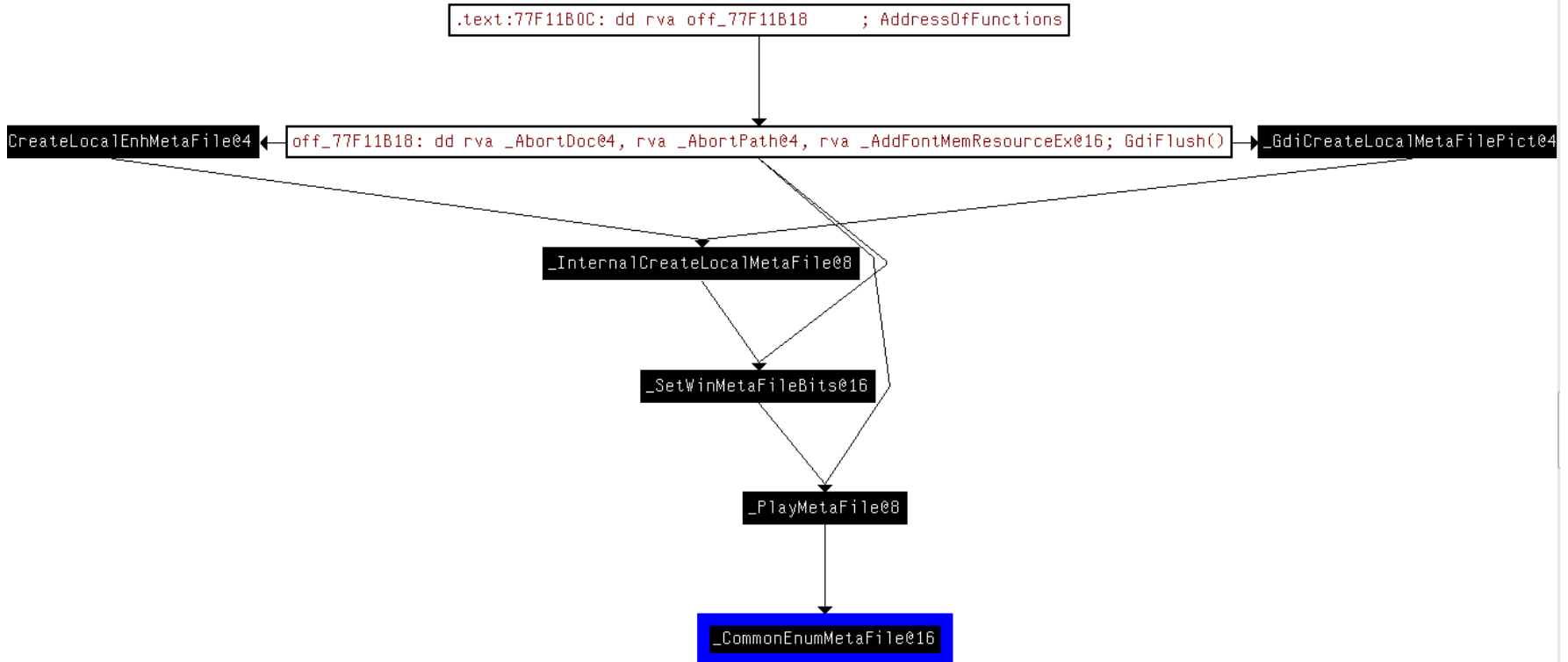
```
text:77F330A3 call eax
```

More Context Helps More

Module: **gdi32.dll**

Function: **CommonEnumMetaFile**

Call Graph: caller **PlayMetaFile**



# WMF -- The Relevant Parts

The screenshot shows a Windows window titled "escape - Windows Picture and Fax Viewer". Inside the window, there is a text box containing the command: `rundll32.exe c:\windows\system32\shimgvw.dll,ImageView_Fullscreen C:\escape\escape.wmf`. Below the command, there is a callout box with the text: "Dynamic Facts: Out of 229,679 instructions executed just to cause the crash ONLY 12 Unique Instructions Are Relevant to the CRASH". To the right of the callout box is an error dialog box titled "Run a DLL as an App" with the following text: "Run a DLL as an App has encountered a problem and needs to close. We are sorry for the inconvenience. If you were in the middle of something, the information you were working on might be lost. Please tell Microsoft about this problem. We have created an error report that you can send to help us improve Run a DLL as an App. We will treat this report as confidential and anonymous. To see what data this error report contains, [click here](#)." The dialog box has two buttons: "Send Error Report" and "Don't Send". The Windows taskbar is visible at the bottom of the window.

## The WMF SETABORTPROC Vulnerability

```
rundll32.exe c:\windows\system32\shimgvw.dll,ImageView_Fullscreen C:\escape\escape.wmf
```

**Dynamic Facts:**  
Out of 229,679 instructions executed just to cause the crash  
**ONLY 12 Unique Instructions Are Relevant to the CRASH**

**Run a DLL as an App**  
Run a DLL as an App has encountered a problem and needs to close. We are sorry for the inconvenience.  
If you were in the middle of something, the information you were working on might be lost.  
**Please tell Microsoft about this problem.**  
We have created an error report that you can send to help us improve Run a DLL as an App. We will treat this report as confidential and anonymous.  
To see what data this error report contains, [click here](#).  
Send Error Report Don't Send

# Conclusions

- Our tools support *interactive binary analysis*, with *Replay*, *Dynamic Taint Analysis*, and *Symbolic Execution*.
- **TREE** runs on top of IDA Pro and supports cross-platform trace collection, taint analysis and replay.
- **CBASS** (based on REIL) enables IR-based architecture-independent symbolic execution and can support both automated and interactive analysis.
- **YOU drive the tools!**



# Where You Can Get TREE

- TREE is open source at:  
<http://code.google.com/p/tree-cbass/>
  - First version of TREE (Taint Analysis) is released
  - Replay is in Progress
  - CBASS is Following
- Contacts:
  - [Li.L.Lixin@gmail.com](mailto:Li.L.Lixin@gmail.com), Project Lead
  - [xingzli@gmail.com](mailto:xingzli@gmail.com), Developer
  - [locvnguy@gmail.com](mailto:locvnguy@gmail.com), Developer
  - [james.just@gmail.com](mailto:james.just@gmail.com), Program Manager

# Acknowledgements

- Thanks to Ilfak Guilfanov and the IDA team for promptly fixing the bugs that we have reported to them and for their suggestions on the GUI integration.
- Thanks to Thomas Dullien and Tim Kornau of the Google Zynamics team for making their latest version of REIL available to us.
- Thanks to numerous reviewers at Battelle Memorial Institute for their feedback

# References

- [1] L. Li and C. Wang. , Dynamic analysis and debugging of binary code for security applications, (to appear) *International Conference on Runtime Verification (RV'13)*. Rennes, France. 2013
- [2] Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network And Distributed System Security Symposium(2008)
- [3] Song, Dawn, et al. "BitBlaze: A new approach to computer security via binary analysis." *Information systems security*. Springer Berlin Heidelberg, 2008. 1-25.
- [4] Clause, James, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework." *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007.
- [5] Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)." In *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 317-331. IEEE, 2010.
- [6] Bhansali, Sanjay, et al. "Framework for instruction-level tracing and analysis of program executions." *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006.
- [7] Dullien, T., Porst, S.: REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In:CanSecWest(2009)
- [8] REIL:URL:[http://www.zynamics.com/binnavi/manual/html/reil language.htm](http://www.zynamics.com/binnavi/manual/html/reil%20language.htm)