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

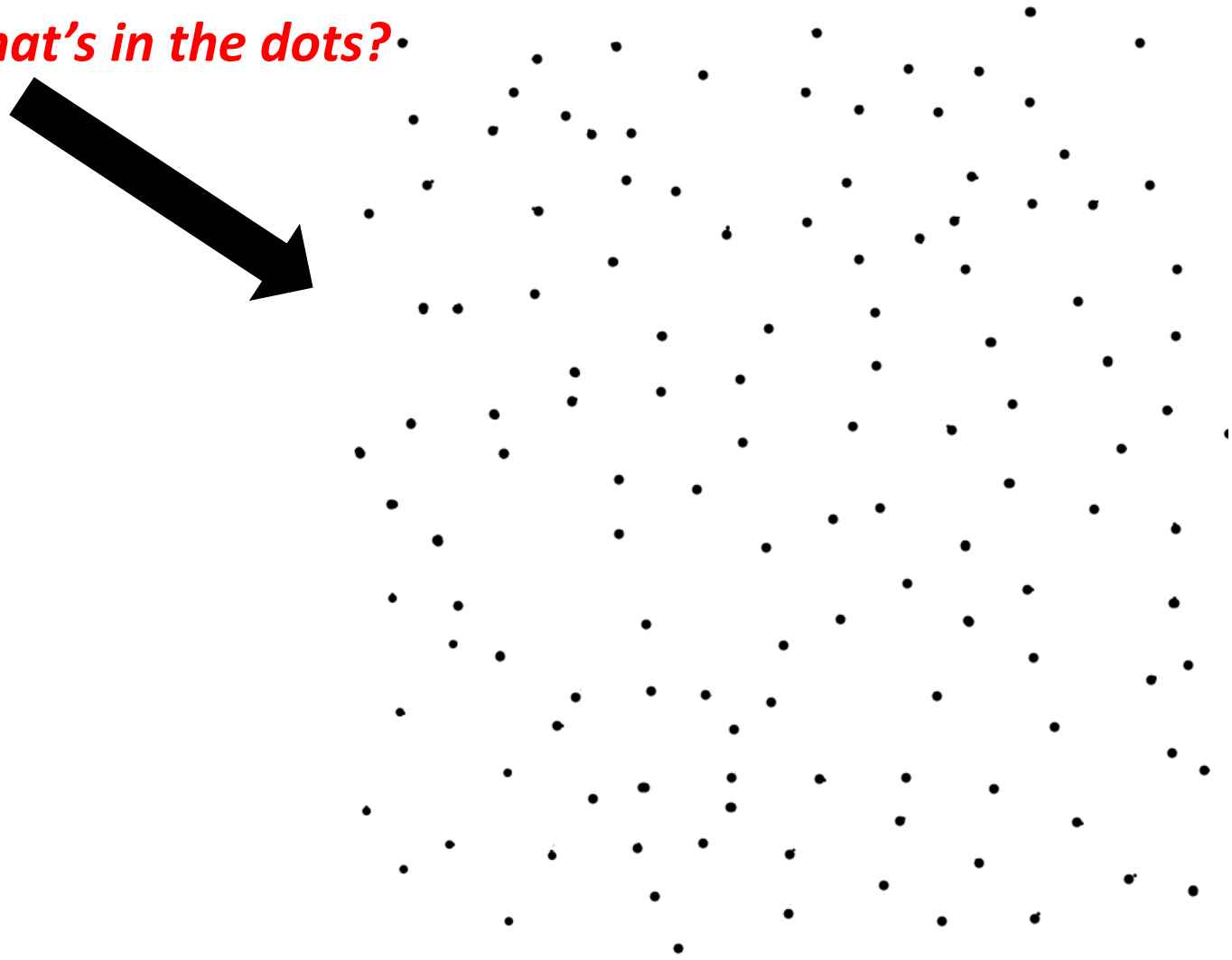
Benefits of Interactive Binary Analysis

- Applicable to many security problems

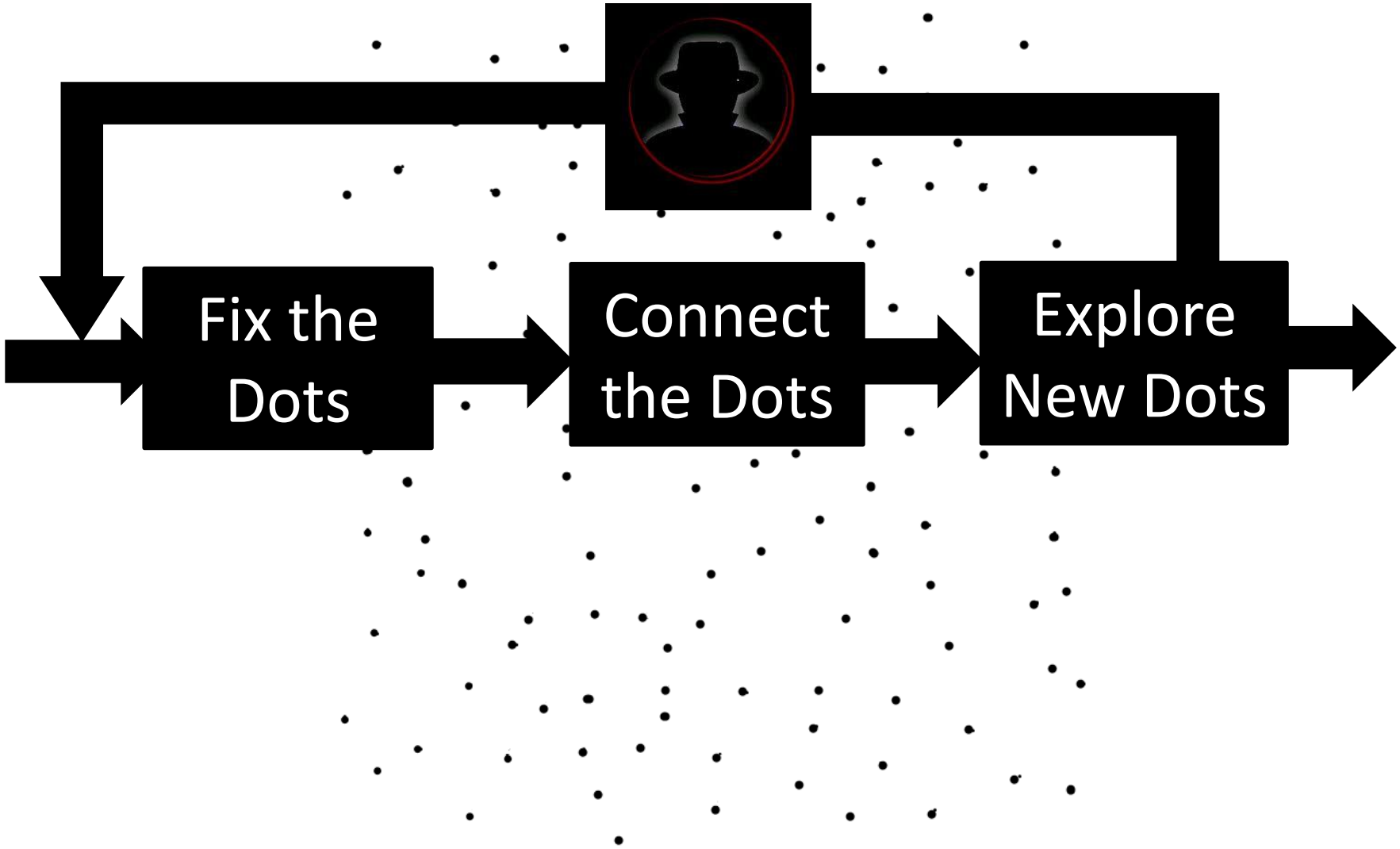
- Our tools increase productivity in:
  - Finding vulnerabilities
  - Analyzing root causes
  - Exploitability and risk assessment

Interactive Analysis Like Connecting Dots

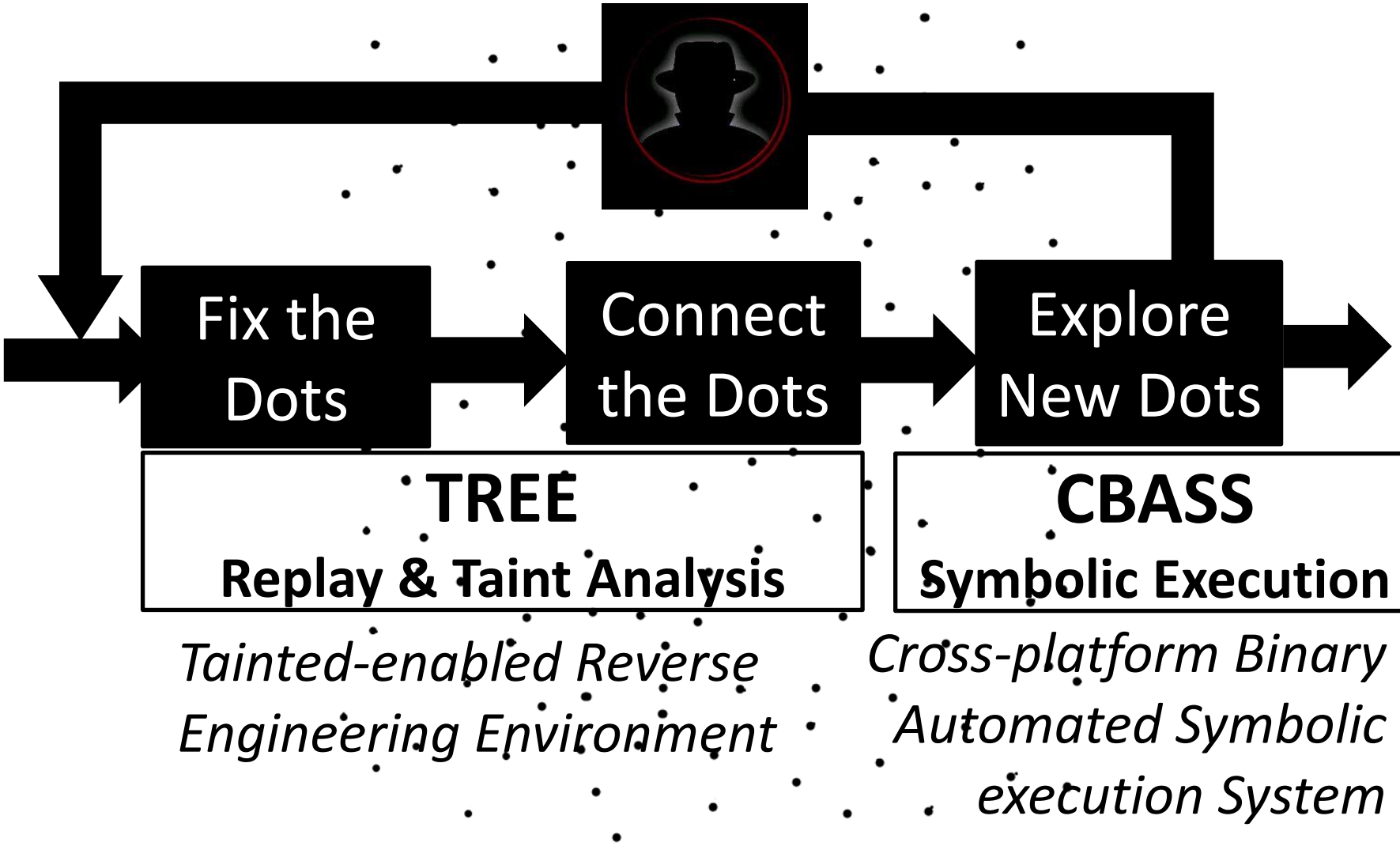
What's in the dots?



Our Tools are Designed to Help



What Do Our Tools Do?



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

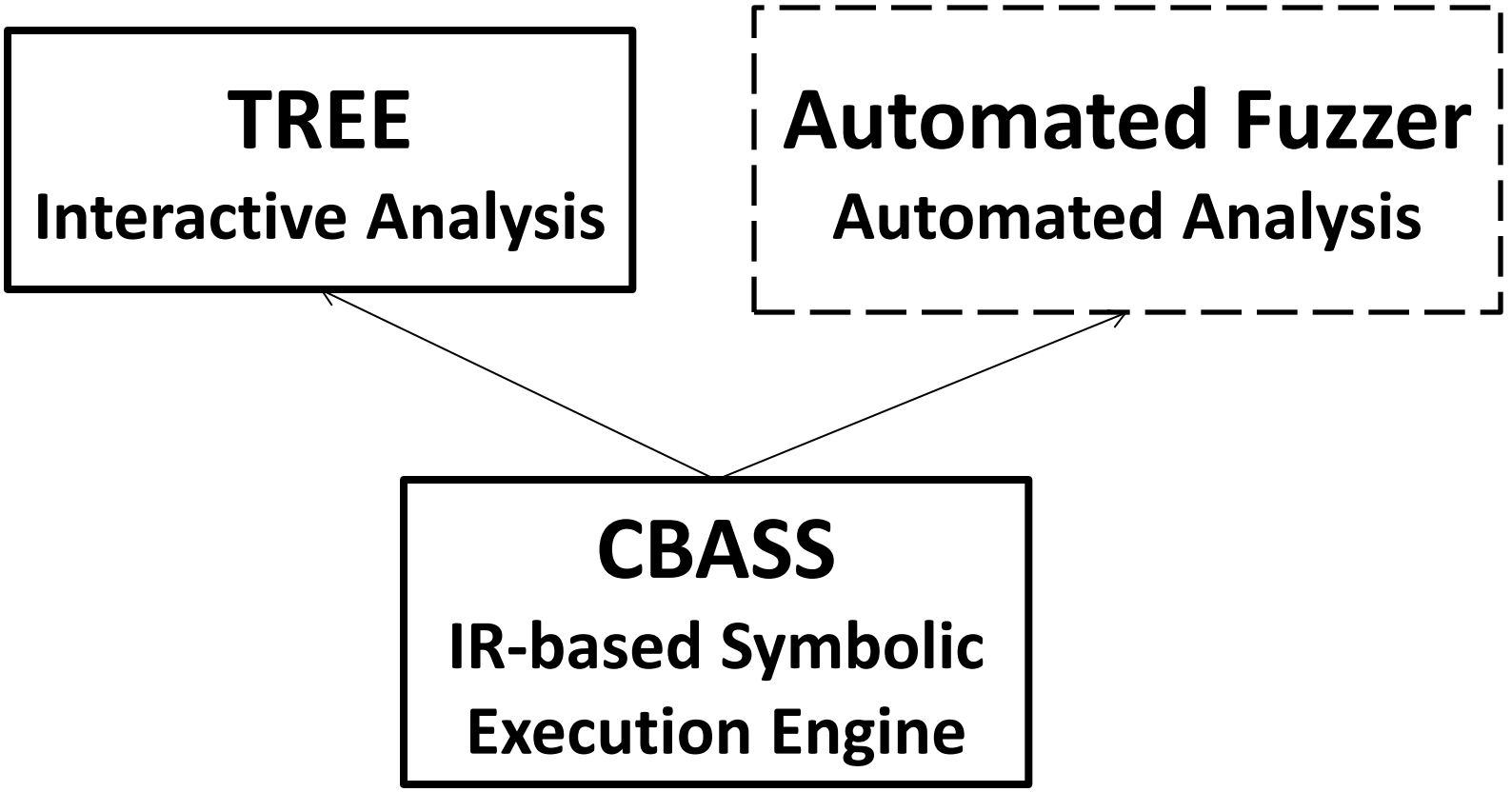
Bringing Proven Research Techniques to Interactive Binary Analysis

- 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

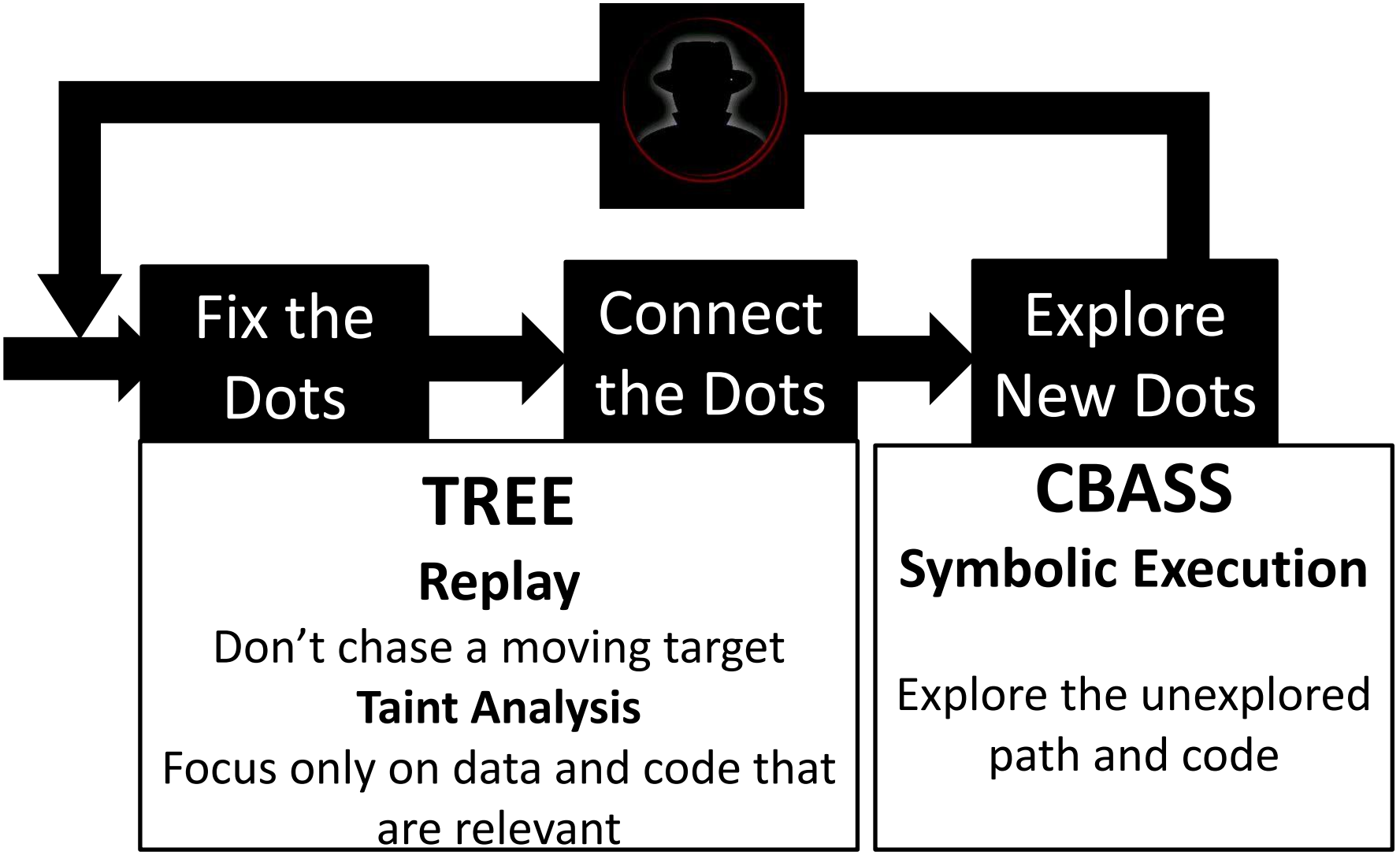
- TREE integrates with IDA Pro now and other mainstream binary analysis environments (later)
- TREE leverages debugging infrastructure to support tracing on multiple platforms
- CBASS uses Intermediate Representation (REIL [6][7])-based approach to support ISA-independent analysis

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

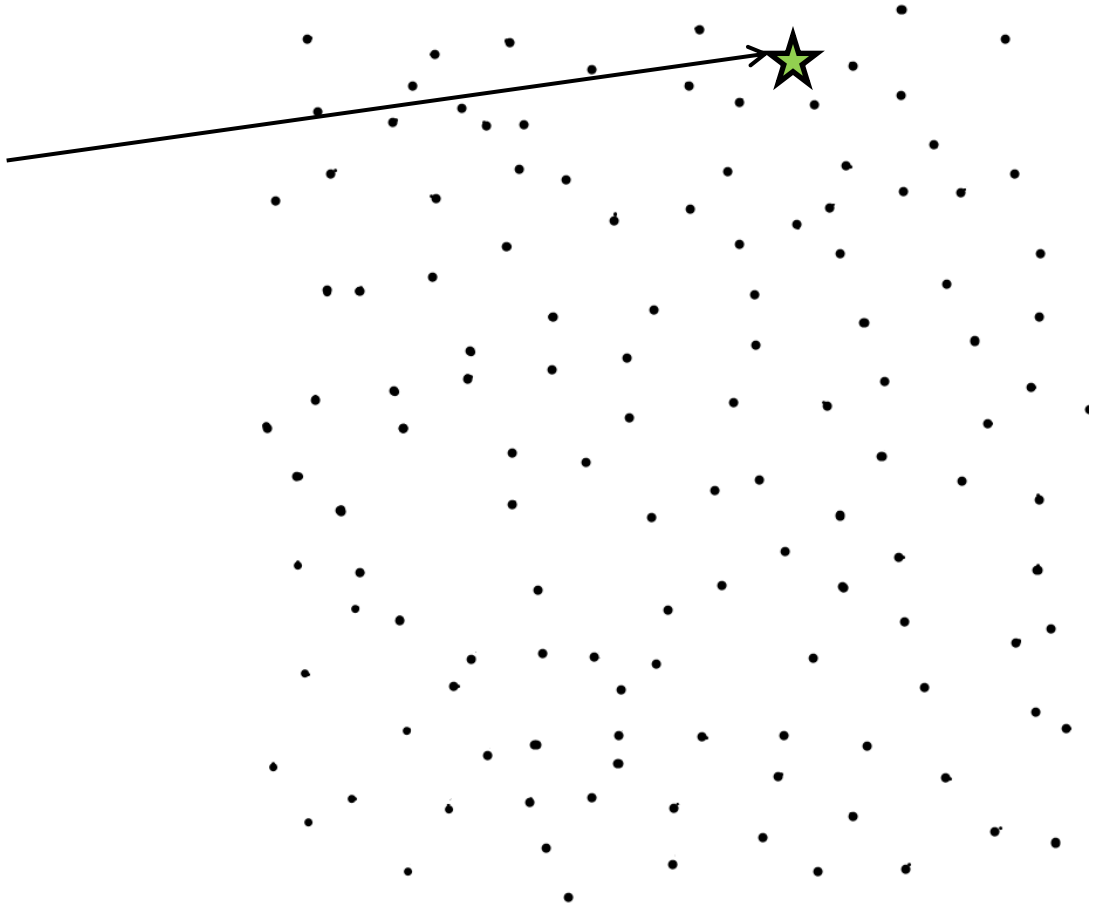
Techniques Help Connect the Dots

- 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

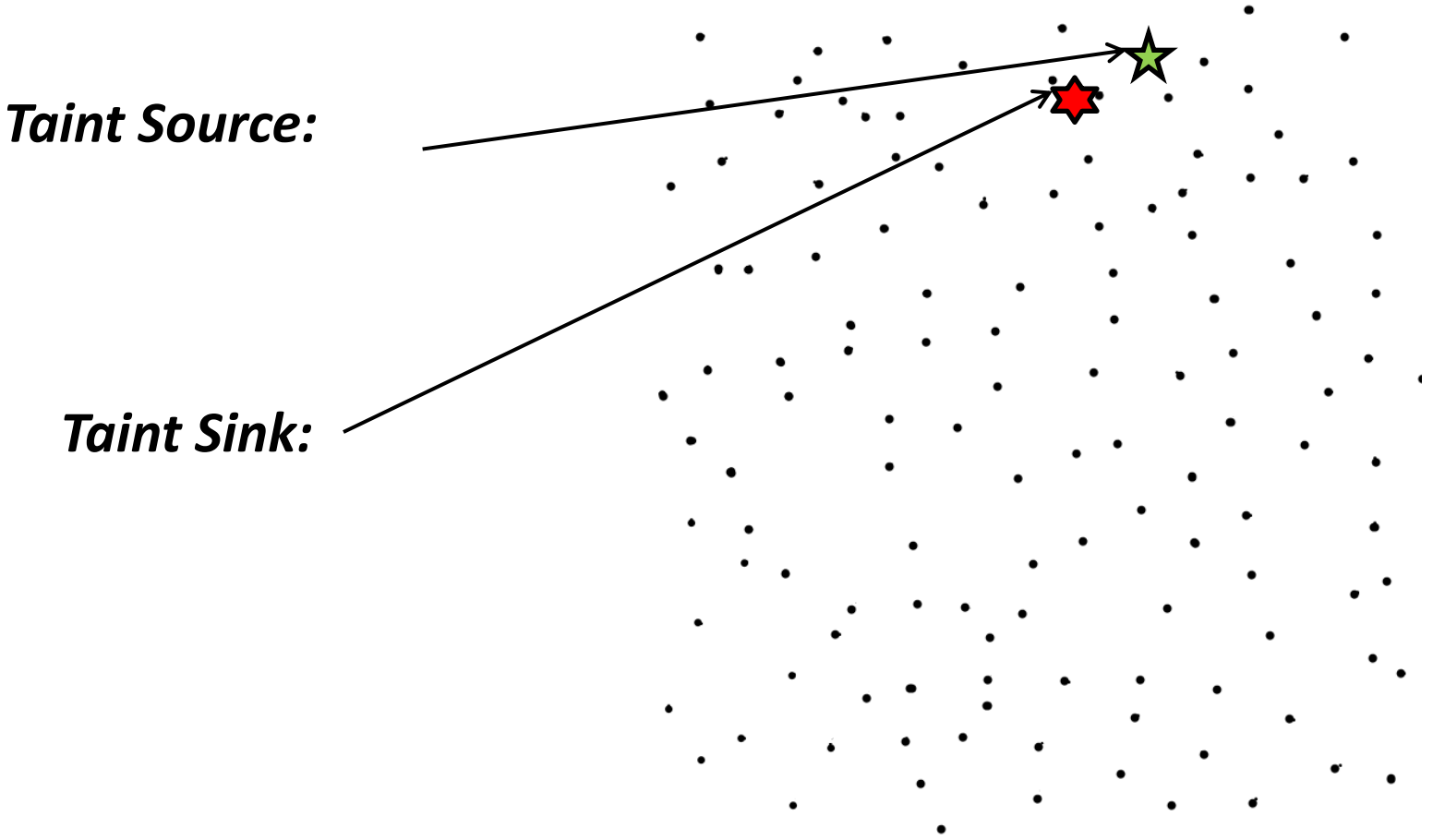
- TREE connects dots -- using taint analysis

Taint Source:



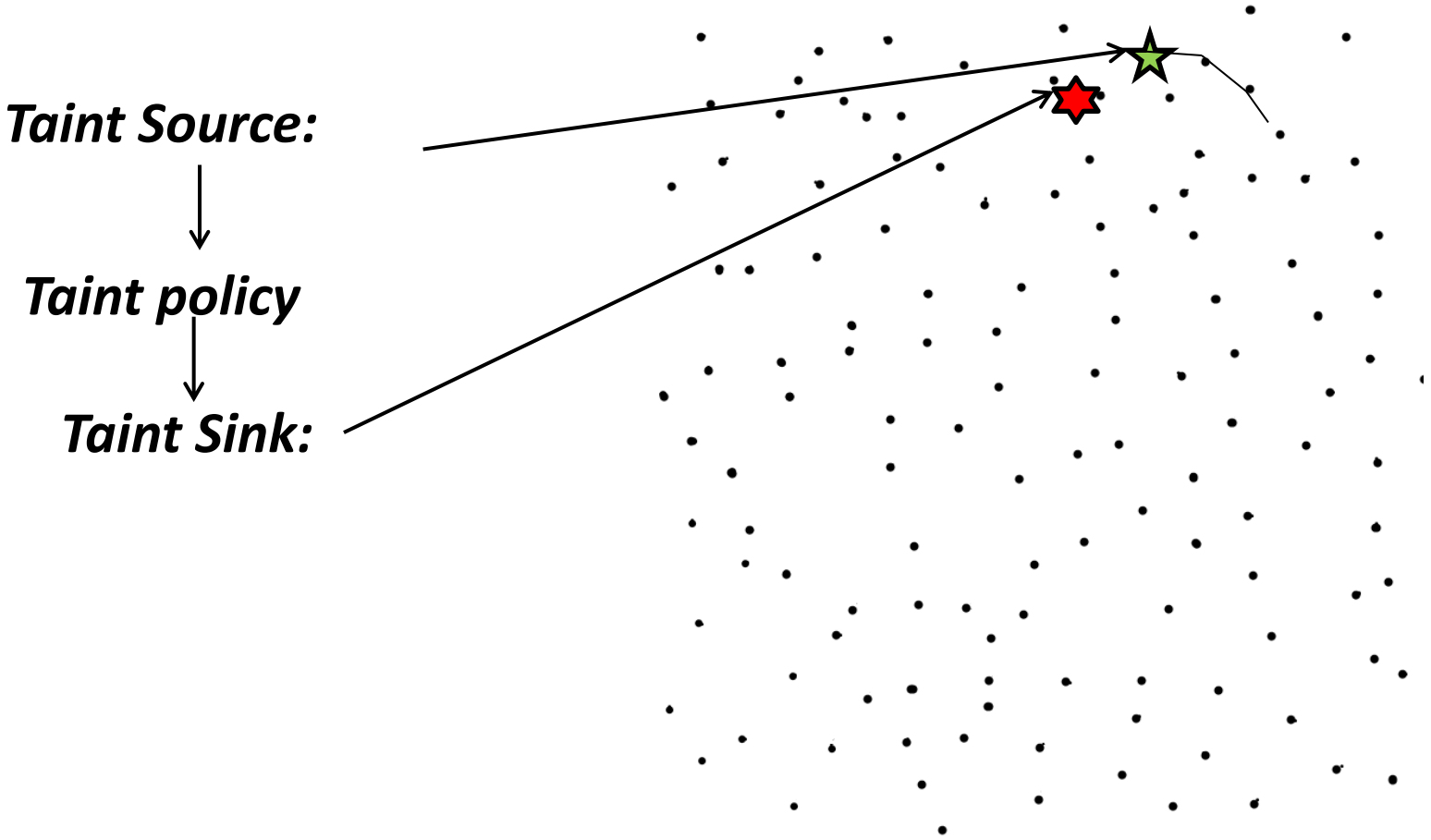
Connect the Dots

- TREE connects dots -- using taint analysis



Connect the Dots

- TREE connects dots -- using taint analysis



Connect the Dots

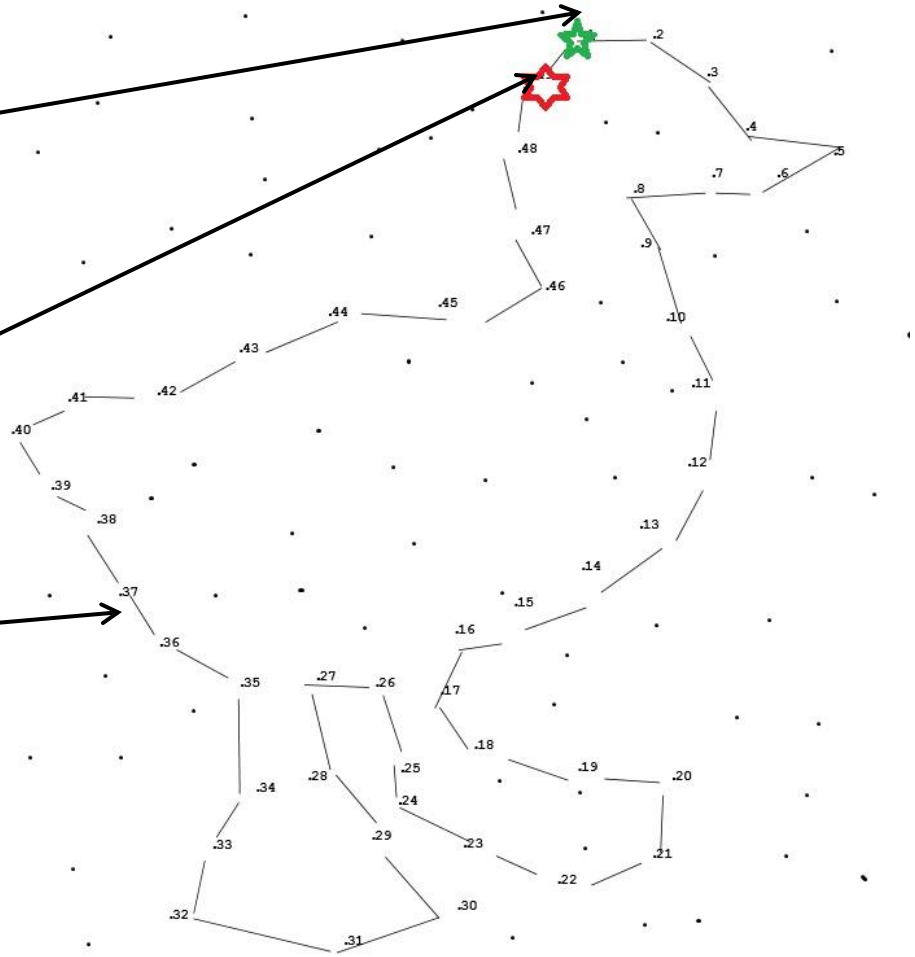
- TREE connects dots -- using taint analysis

Taint Source:

Taint policy

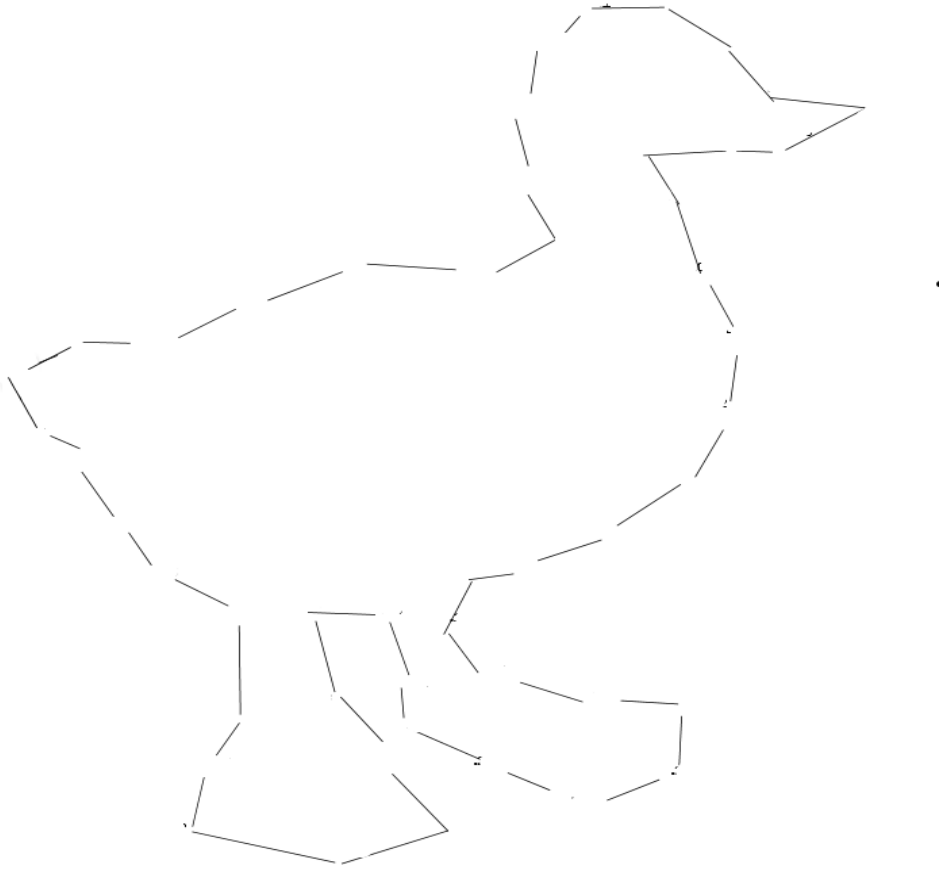
Taint Sink:

- **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

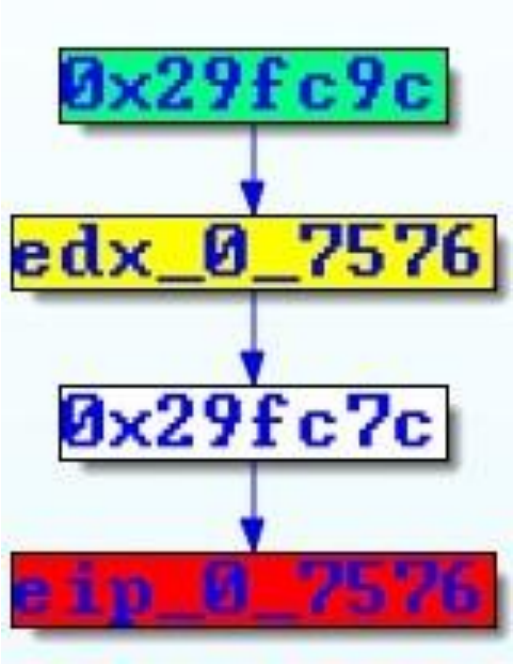
↓

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

↓

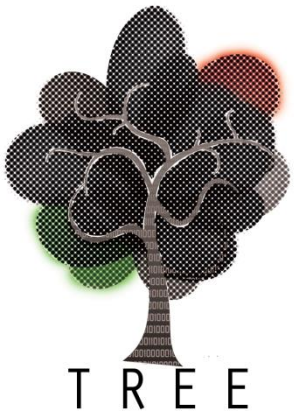
retl

The Taint Graph



What You Connect is What You Get

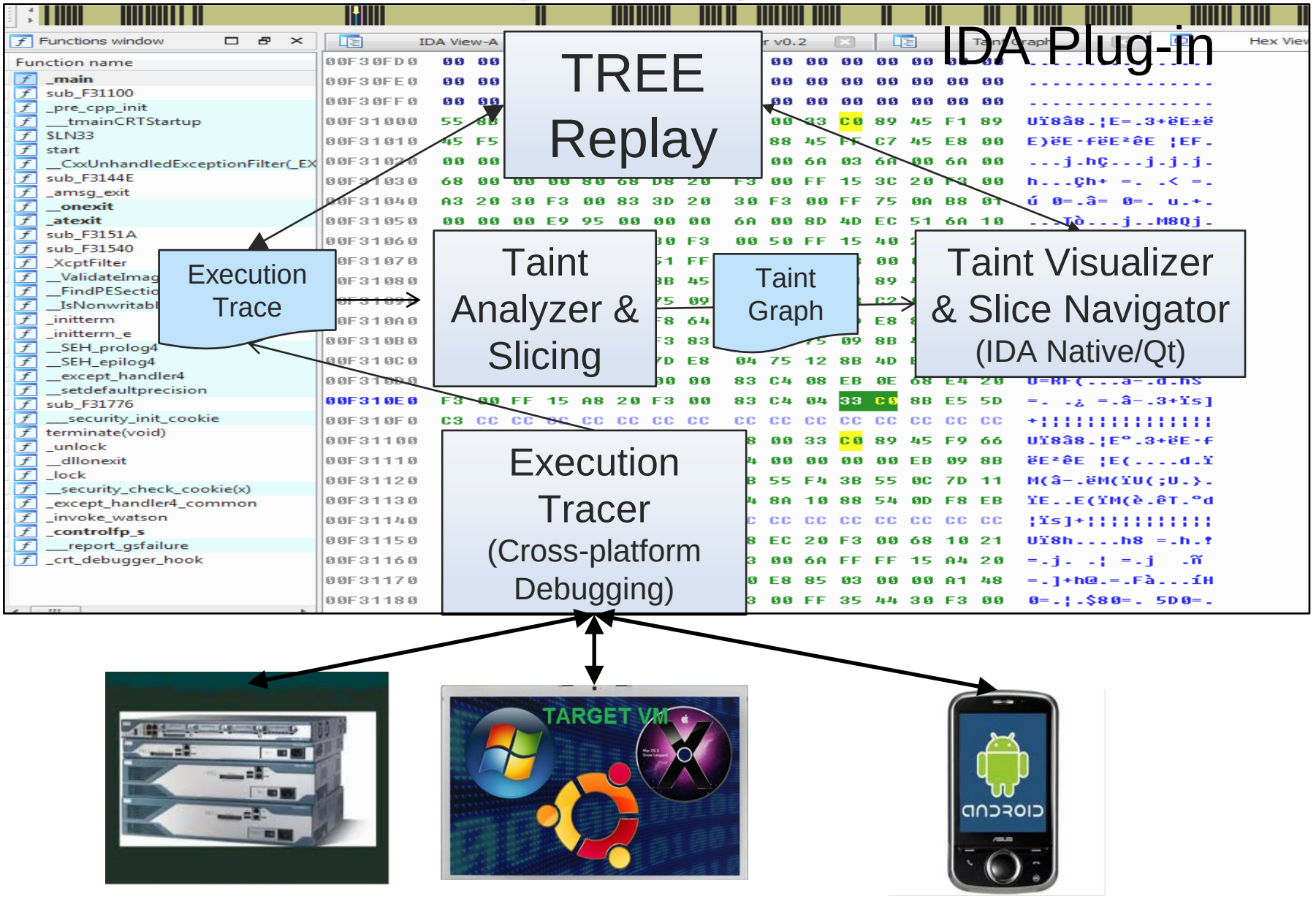
- Dots can be connected in different ways
  - Data dependency
  - Address dependency
  - Branch conditions
  - Loop counter
- Connect dots in different taint policies



TREE

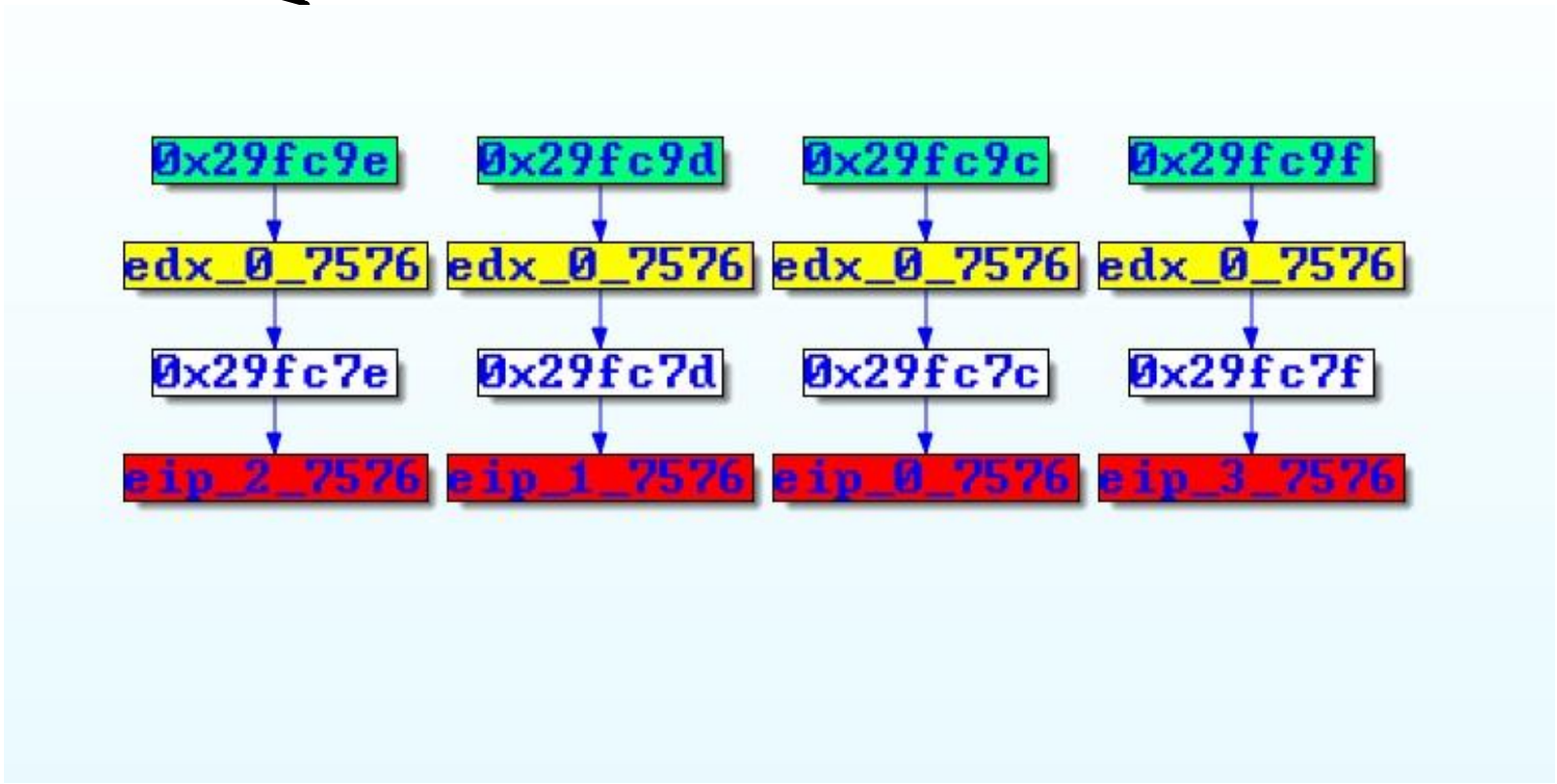
TAINT-ENABLED REVERSE ENGINEERING ENVIRONMENT

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

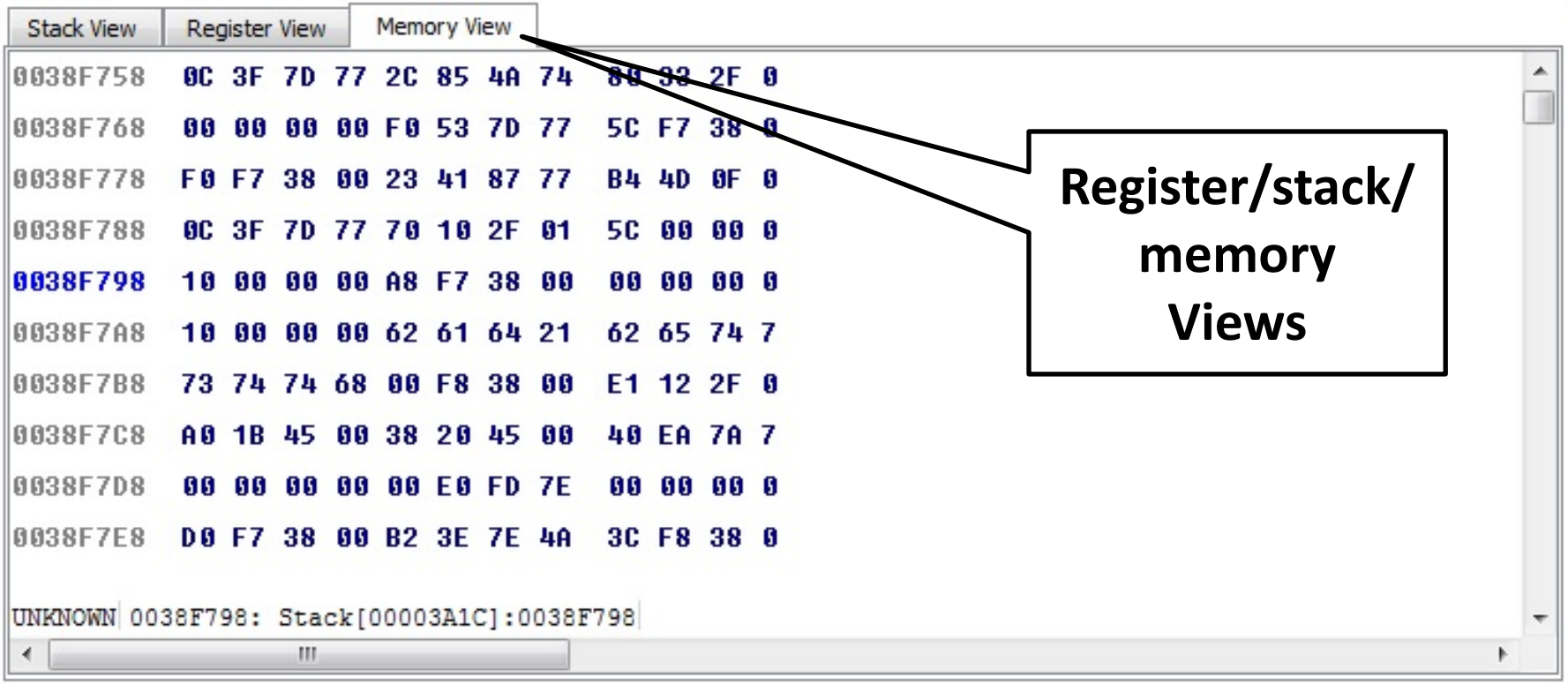
UUID	Type	Name	Start Sequence	End Sequence	formation Instru	Child C	Child D
60	register	eip_3_14876	0x11c		retl		52
59	register	eip_2_14876	0x11c		retl		50
58	register	eip_1_14876	0x11c		retl		48
57	register	eip_0_14876	0x11c		retl		46
52	memory	0x38f79b	0x112		movb %dl, -...		51
51	register	edx_0_14876	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

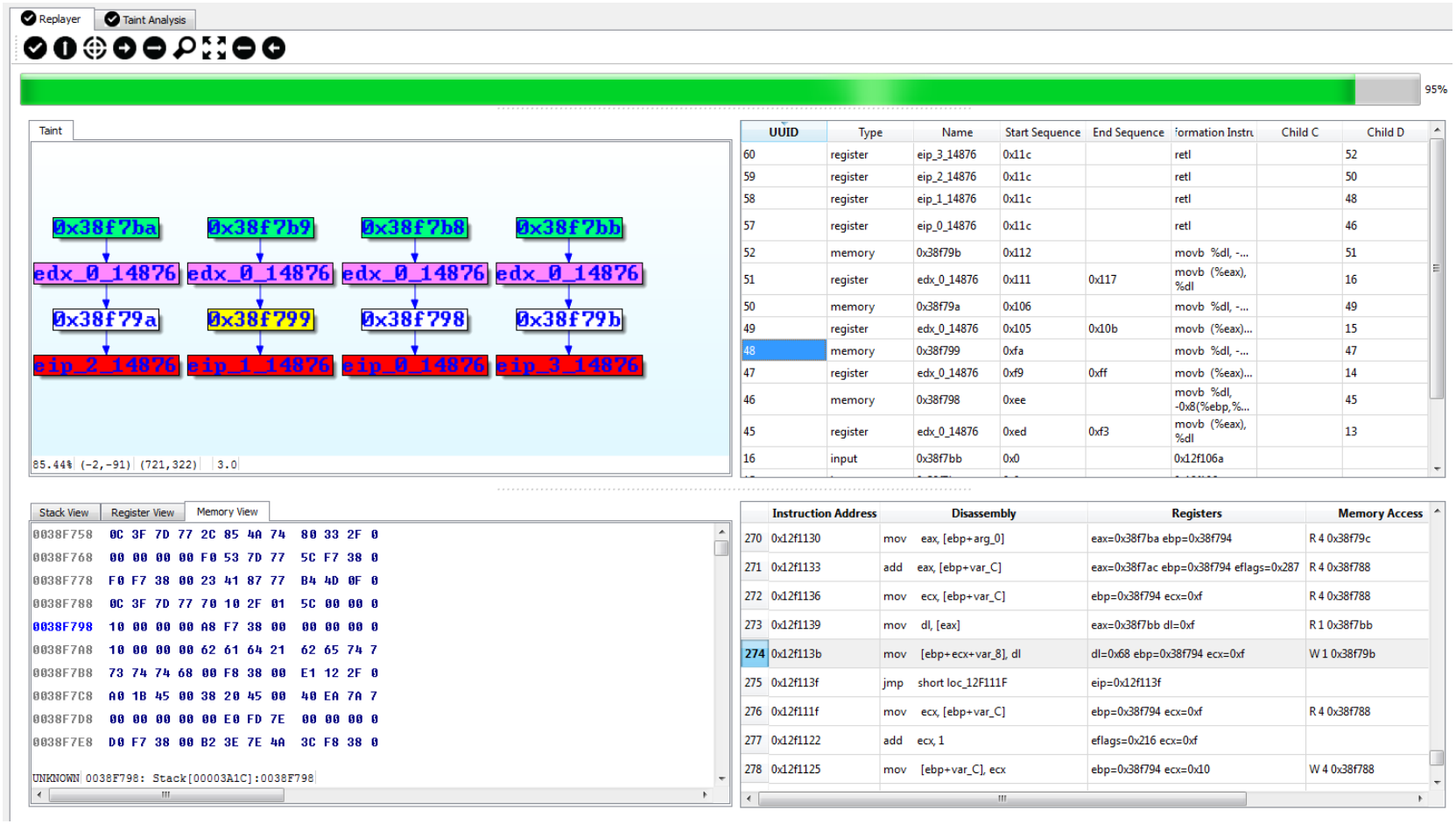
	Instruction Address	Disassembly	Registers	Memory Access
270	0x12f1130	mov eax, [ebp+arg_0]	eax=0x38f7ba ebp=0x38f794	R 4 0x38f79c
271	0x12f1133	add eax, [ebp+var_C]	eax=0x38f7ac ebp=0x38f794 eflags=0x287	R 4 0x38f788
272	0x12f1136	mov ecx, [ebp+var_C]	ebp=0x38f794 ecx=0xf	R 4 0x38f788
273	0x12f1139	mov 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



TREE: The Front-end of Our Interactive Analysis System

Replay is focal point of user interaction



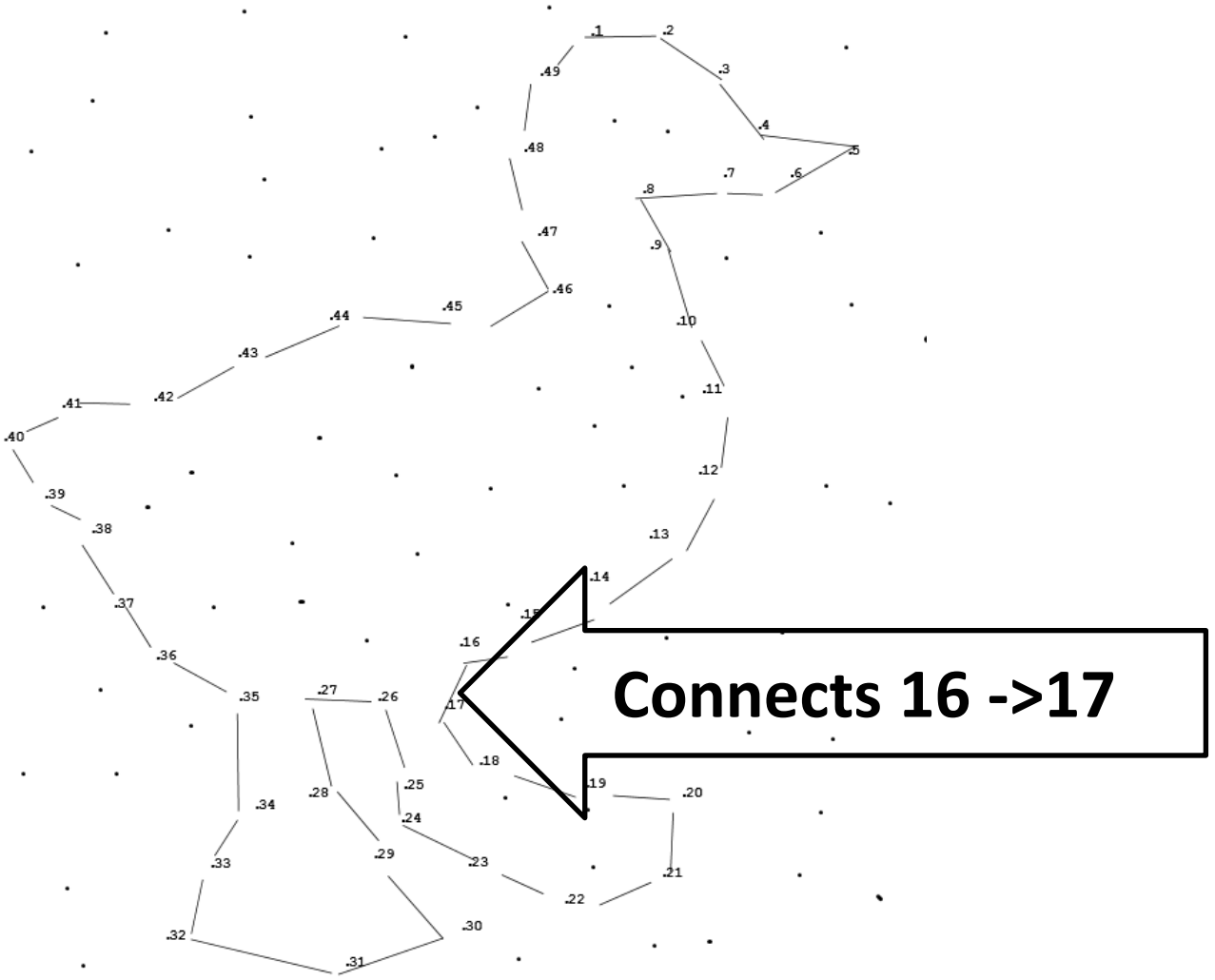
Tree Demo

Using TREE to Analyze a Crash

Our Tools Support

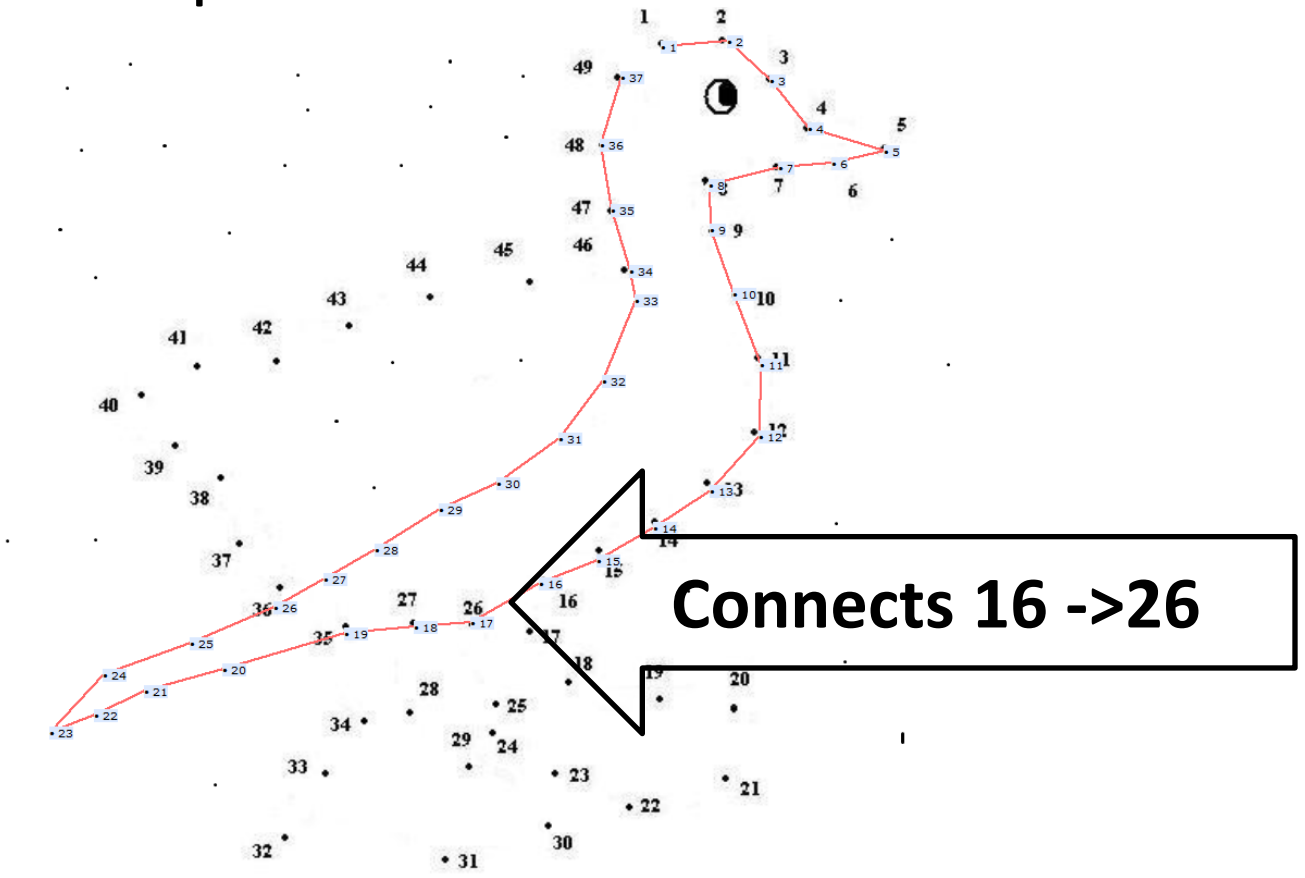
Exploring New Dots

A Key Branch Point for a Duck



The Path for a …

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



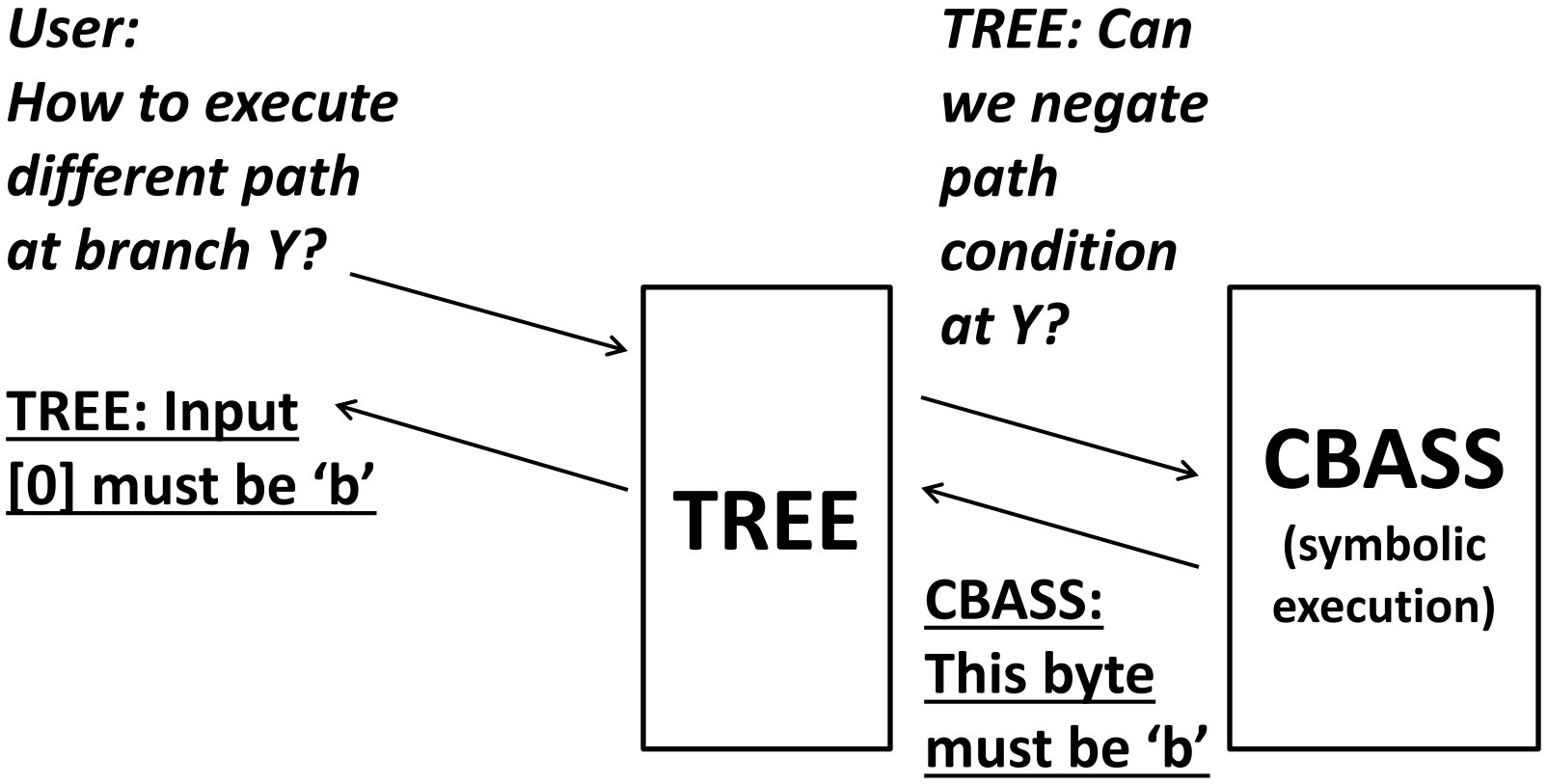
Explore New Dots

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

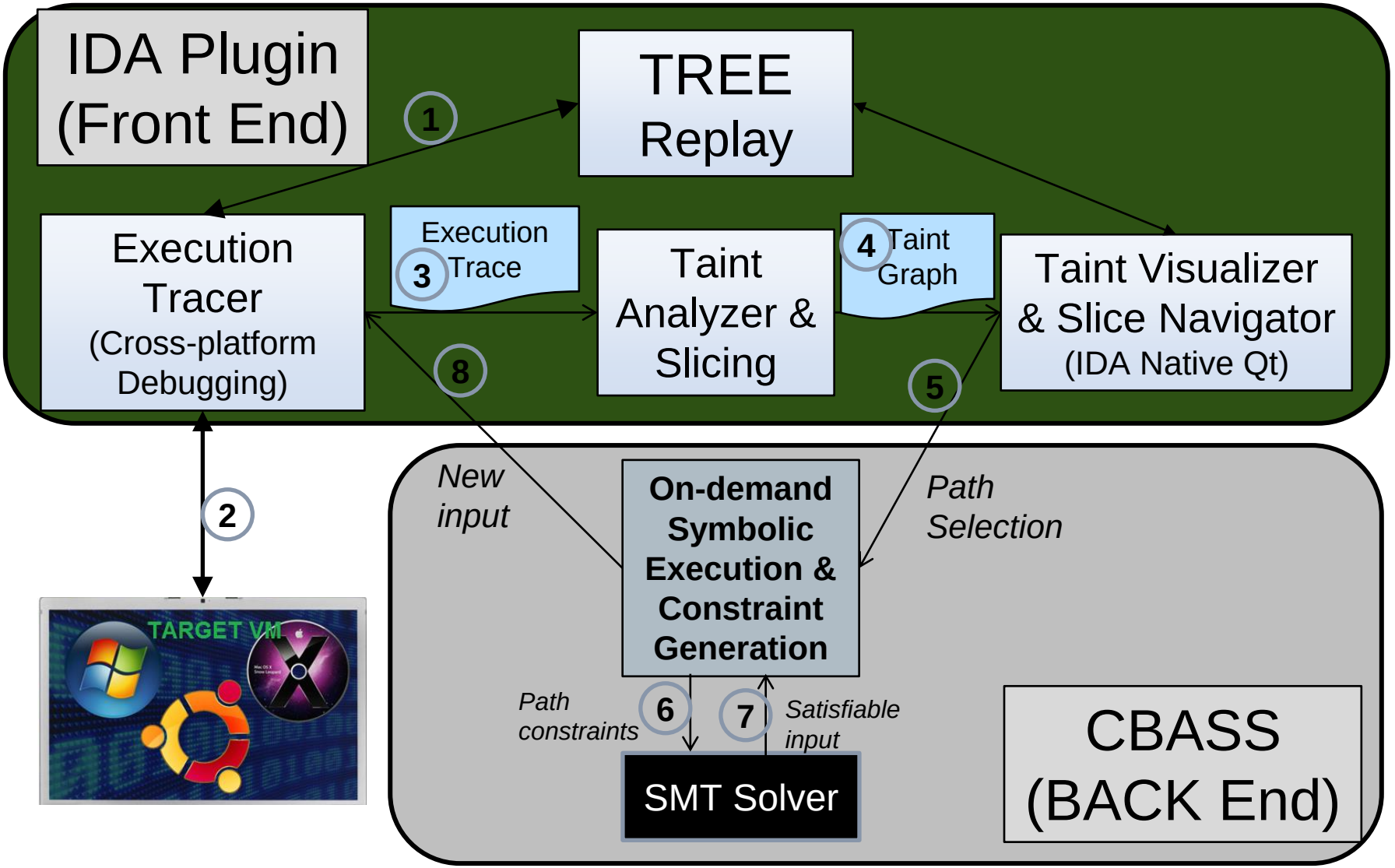
```
//INPUT
ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);
……
//PATH CONDITION
if(sBigBuf[0]=='b') iCount++;
if(sBigBuf[1]=='a') iCount++;
if(sBigBuf[2]=='d') iCount++;
if(sBigBuf[3]=='!') iCount++;
if(iCount==4) // “bad!” path
        StackOVflow(sBigBuf,dwBytesRead) ?
Else // “Good” path
      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

Explore New Dots

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



Explore New Dots Demo



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

***//*INPUT**

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

***//*INPUT TRANSFORMATION**

……

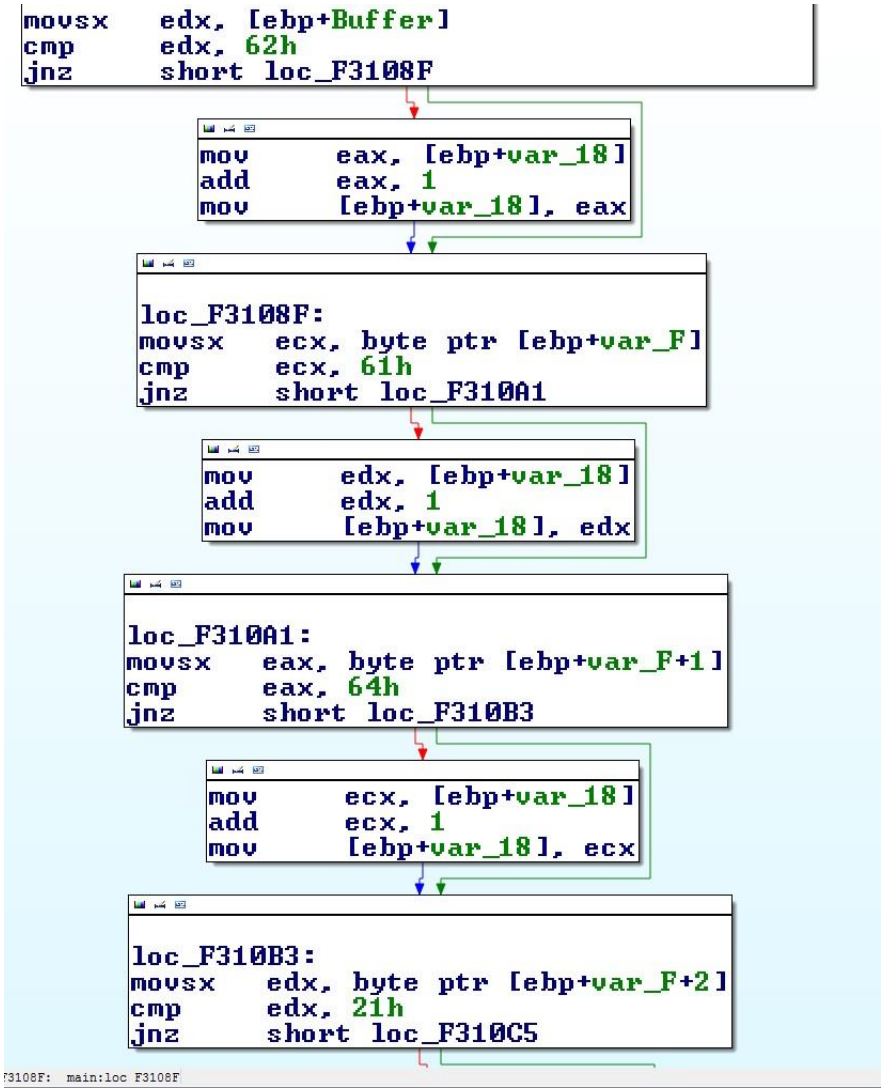
***//*PATH CONDITION**

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**

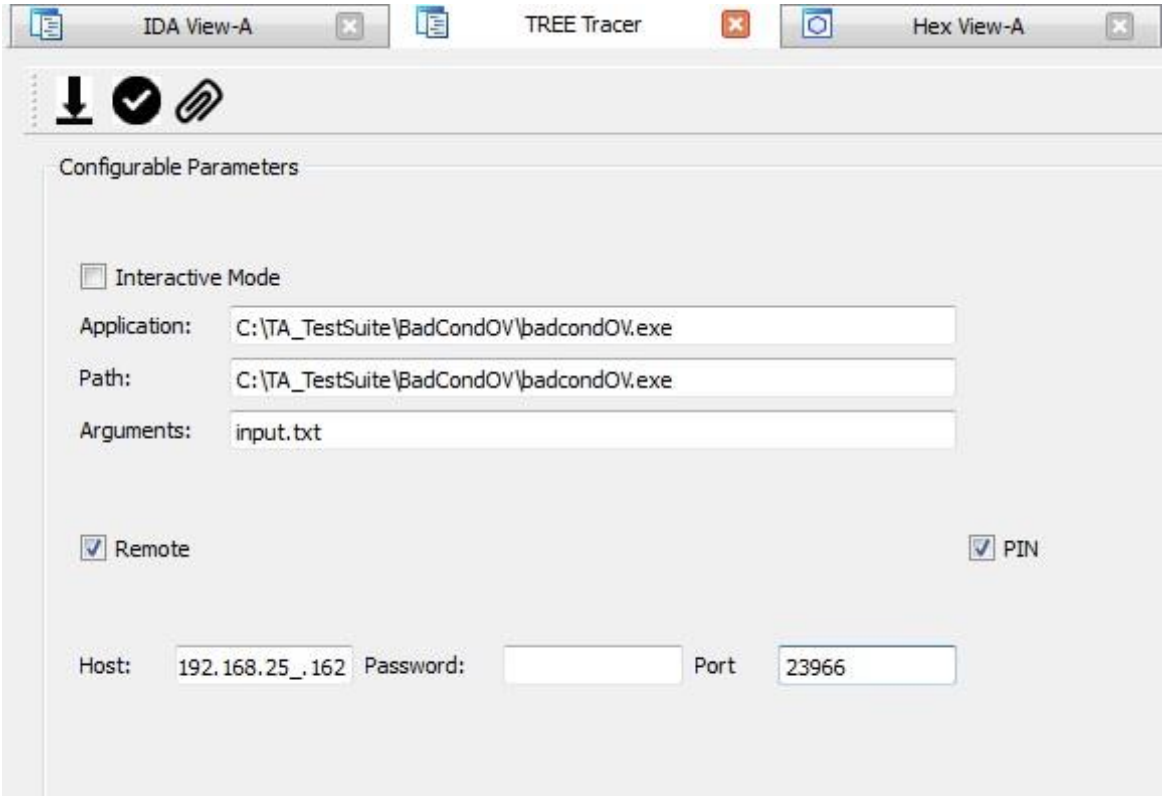
StackOVflow(sBigBuf,dwBytesRead)
else
printf(“Good!”);

Branch Conditions In Disassembly



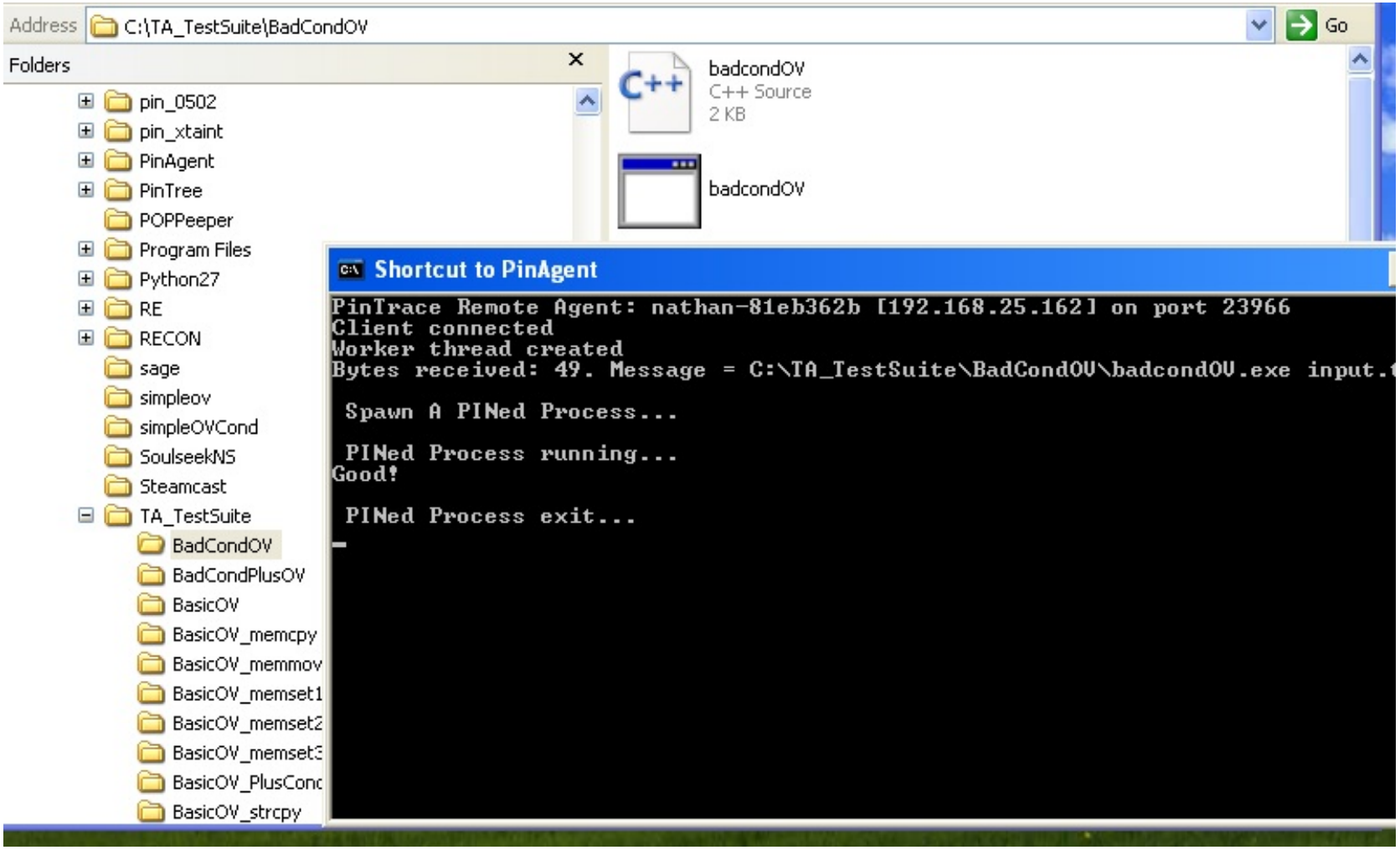
① TREE Pin Trace

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

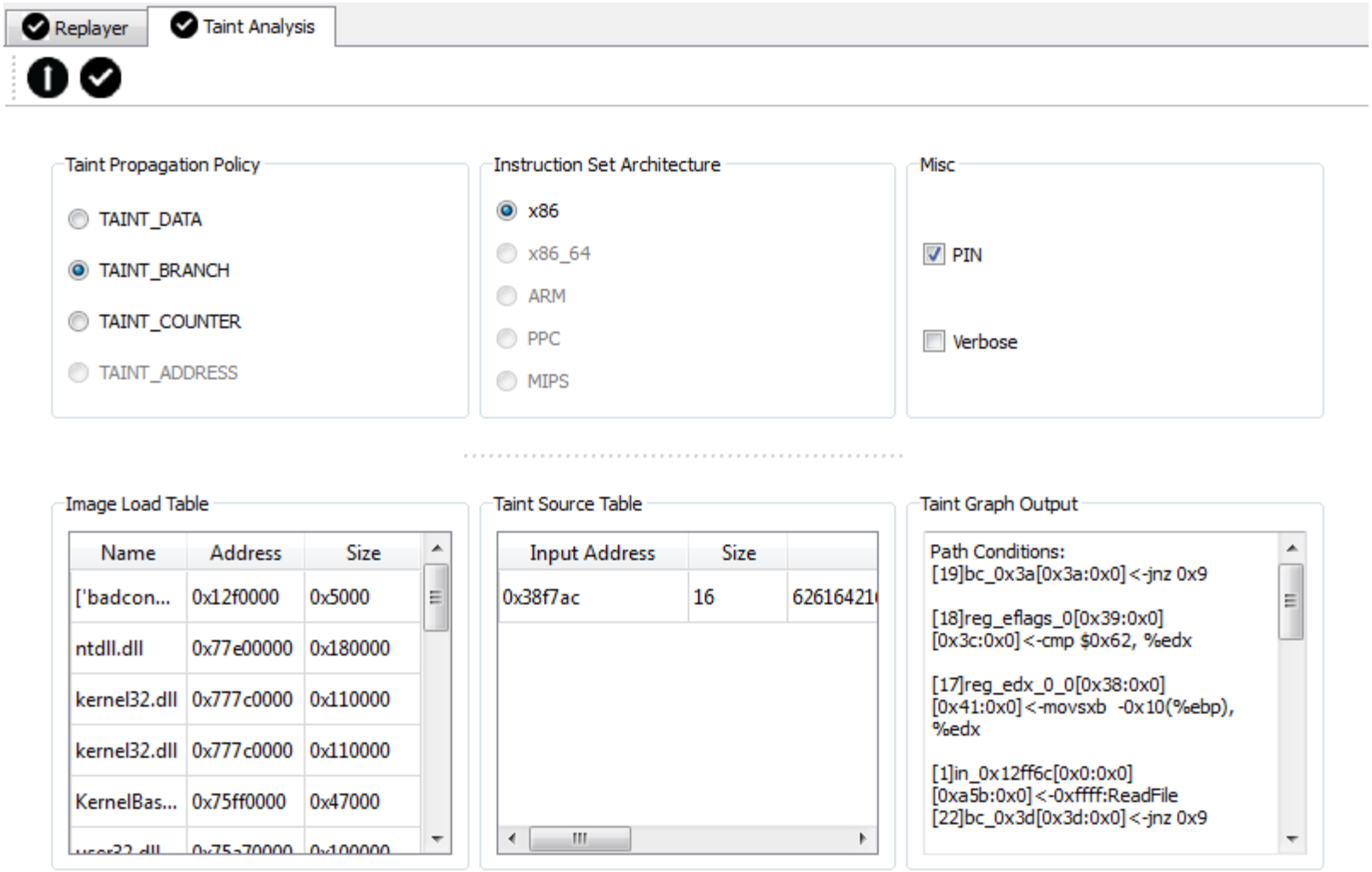


②TREE Console: Trace Generation

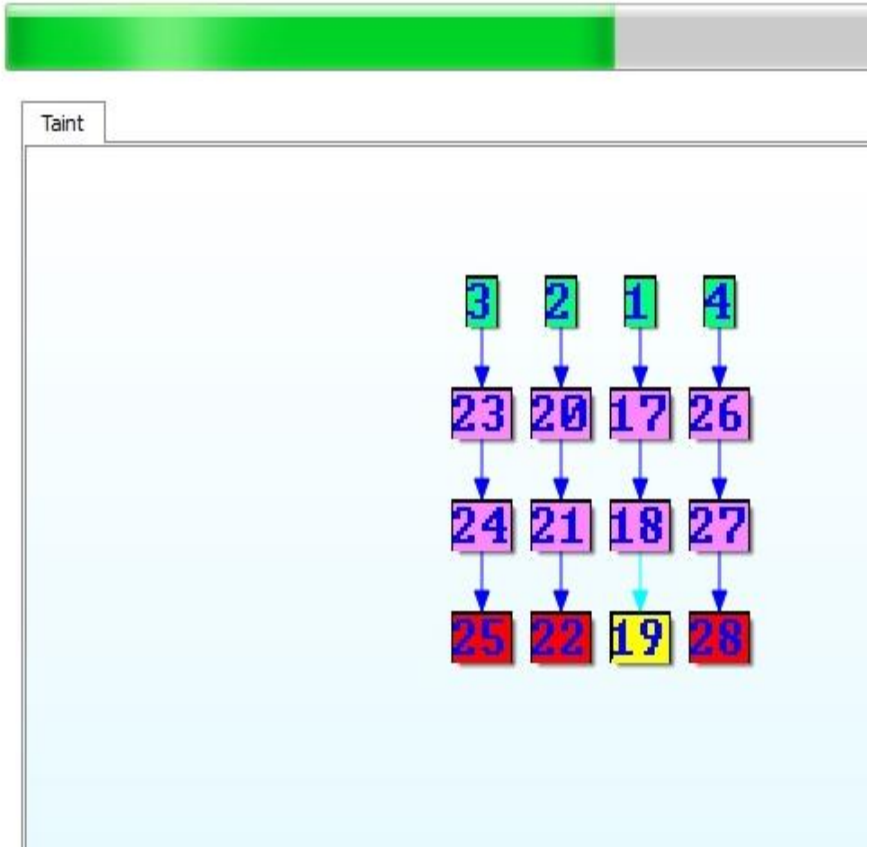
PINAgent: Connects TREE with PIN tracer



③ TREE: Taint Analysis Configuration

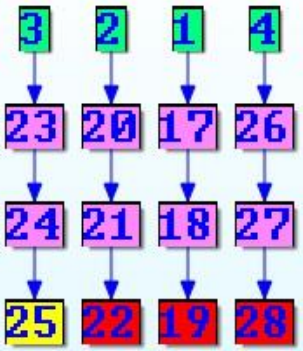


④ TREE: Branch Taint Graph



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

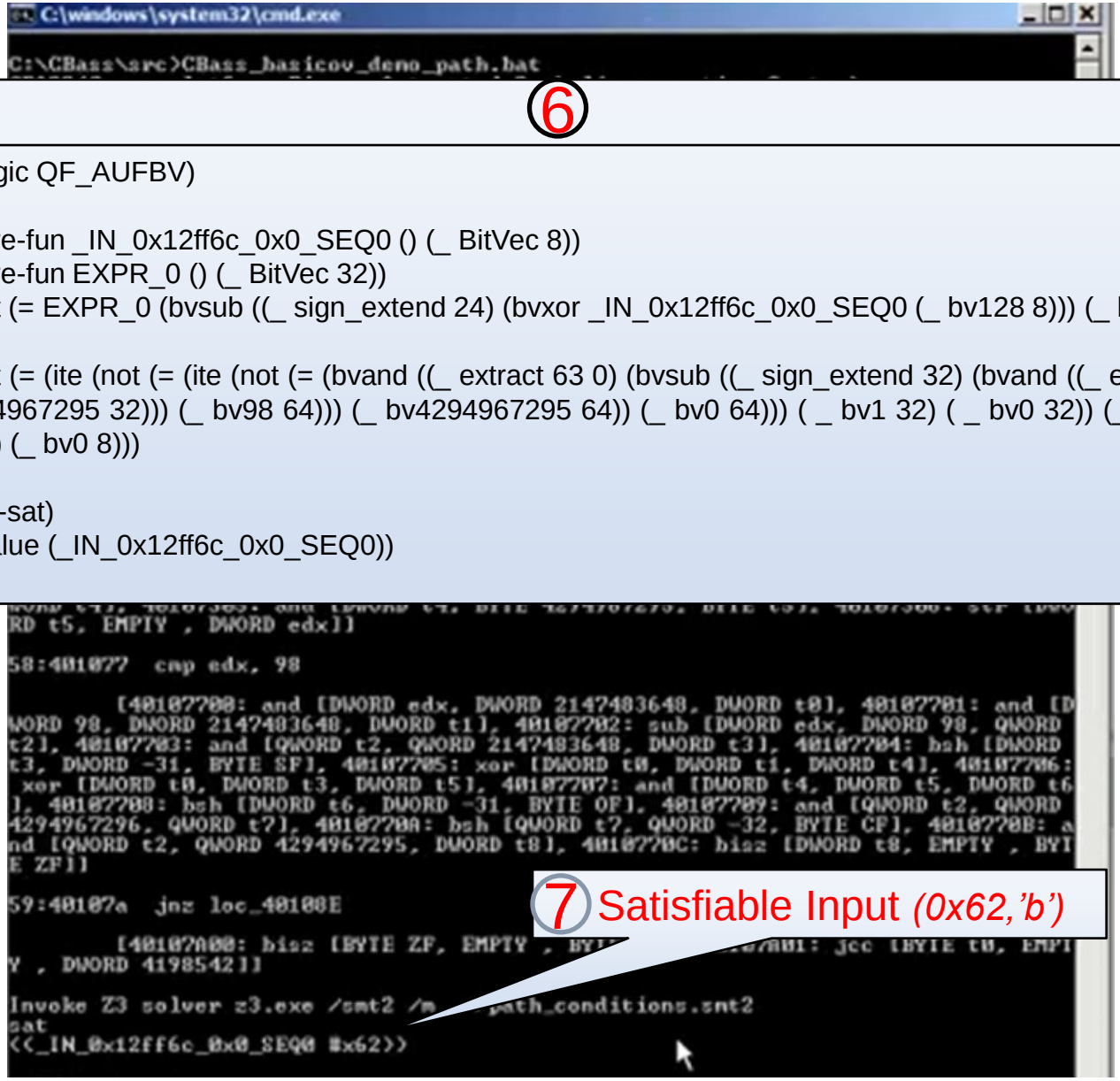
```
Connecting to CBASS server at 127.0.0.1:8888
Query on branch condition of: [19]bc_0x3a
[jnz 0x9]
Result: Offset=0,Value=98
Connecting to CBASS server at 127.0.0.1:8888
Query on branch condition of: [22]bc_0x3d
[jnz 0x9]
Result: Offset=1,Value=97
Connecting to CBASS server at 127.0.0.1:8888
Query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

‘b’

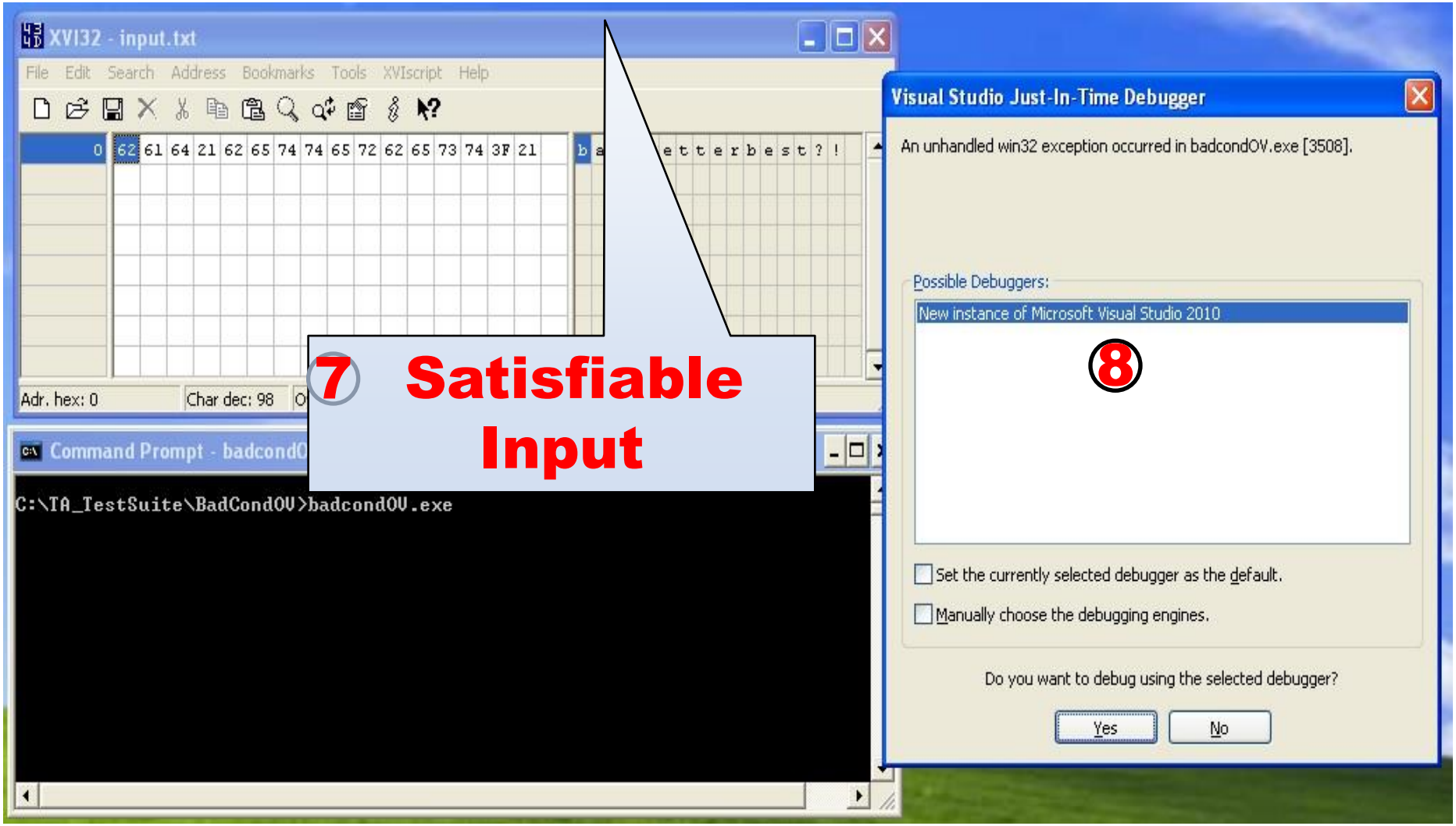
‘a’

‘d’

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



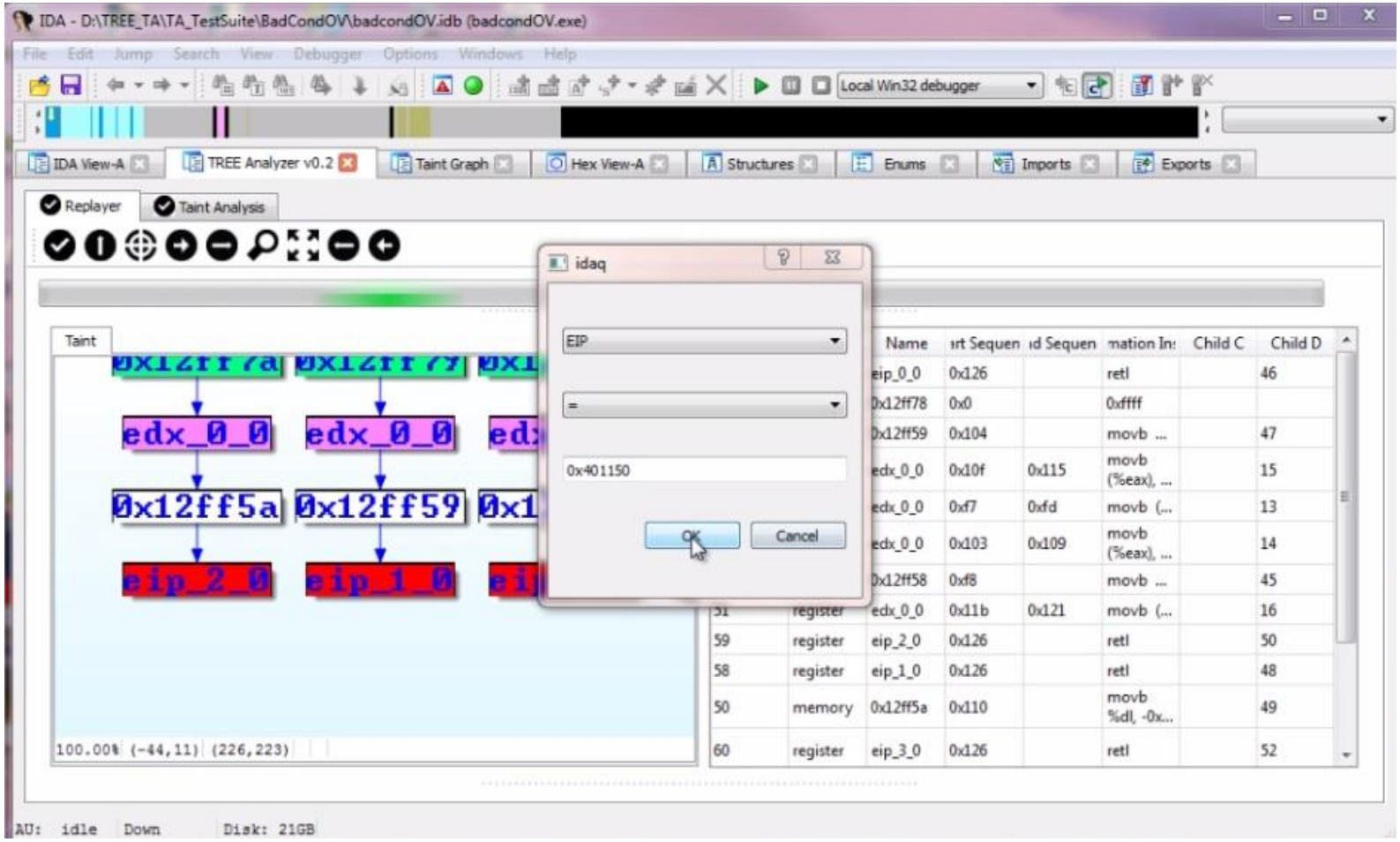
⑧ TREE: Re-execute with "Satisfiable" Input



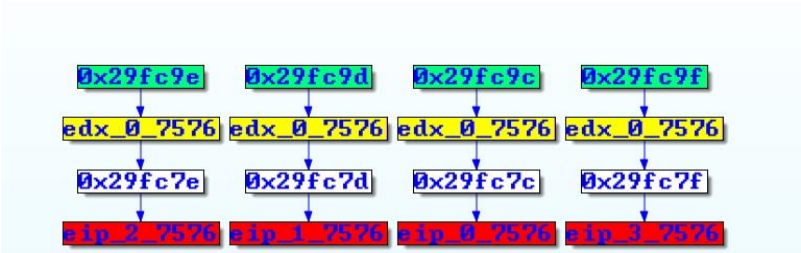
Task 2: Own the Execution
Assume Payload at 0x401150

```
.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

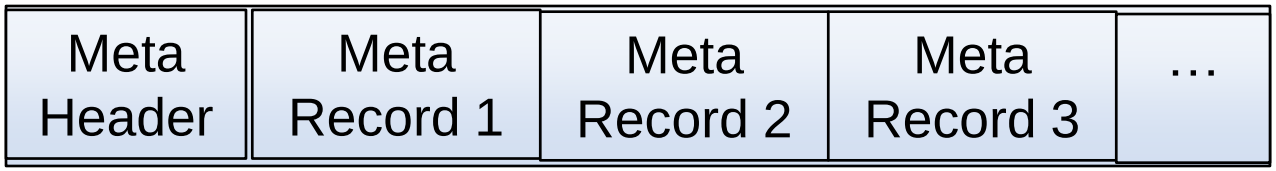
Target Vulnerability	Vulnerability Name	Target Application Mode	Target OS
CVE-2005-4560	Windows WMF	User Mode	Windows
CVE-2207-0038	ANI Vulnerability	User Mode	Windows
OSVDB-2939	AudioCoder Vulnerability	User Mode	Windows
CVE-2011-1985	Win32k Kernel Null Pointer De-reference	Kernel Mode	Windows
CVE-2004-0557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

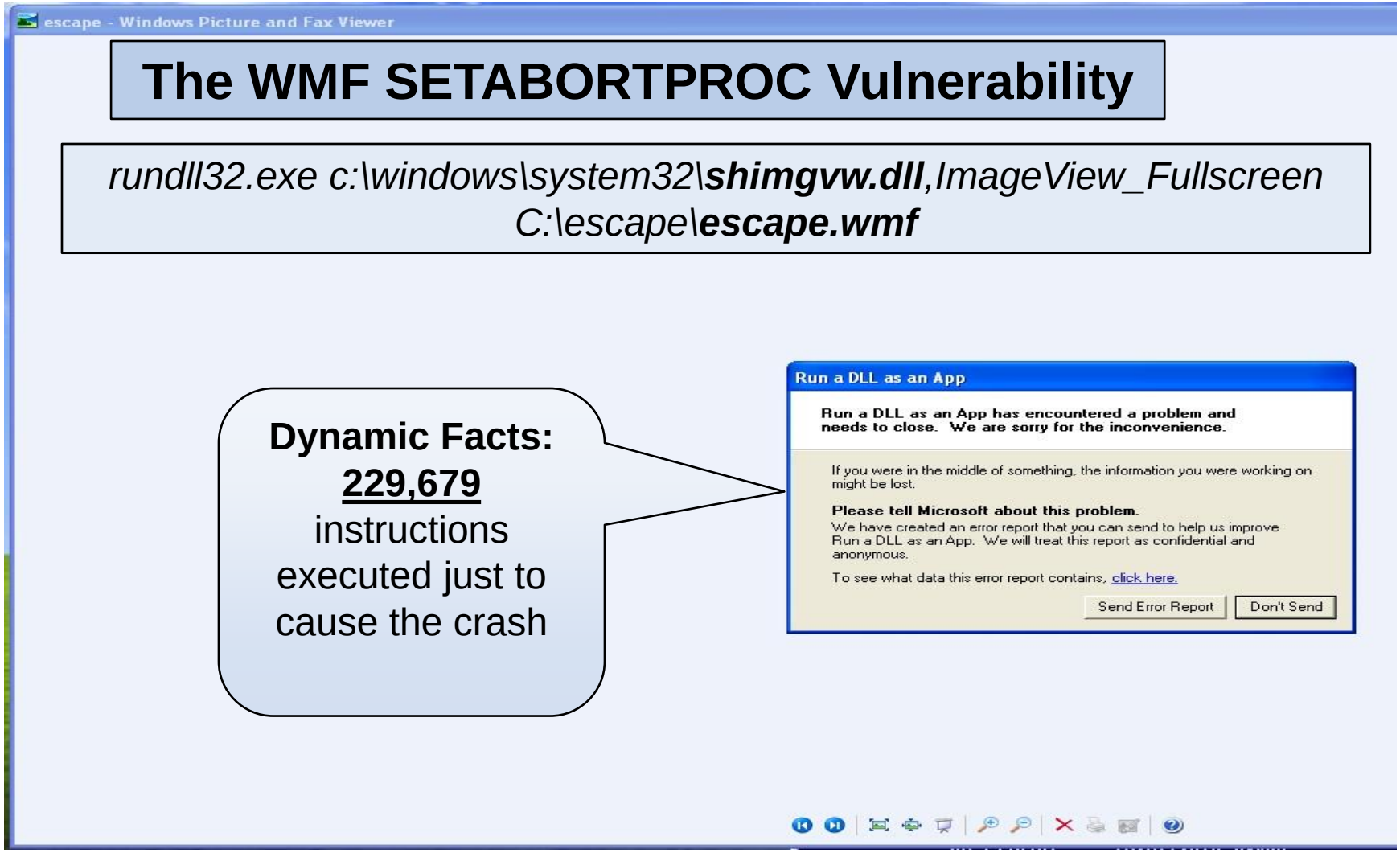
WMF Format

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

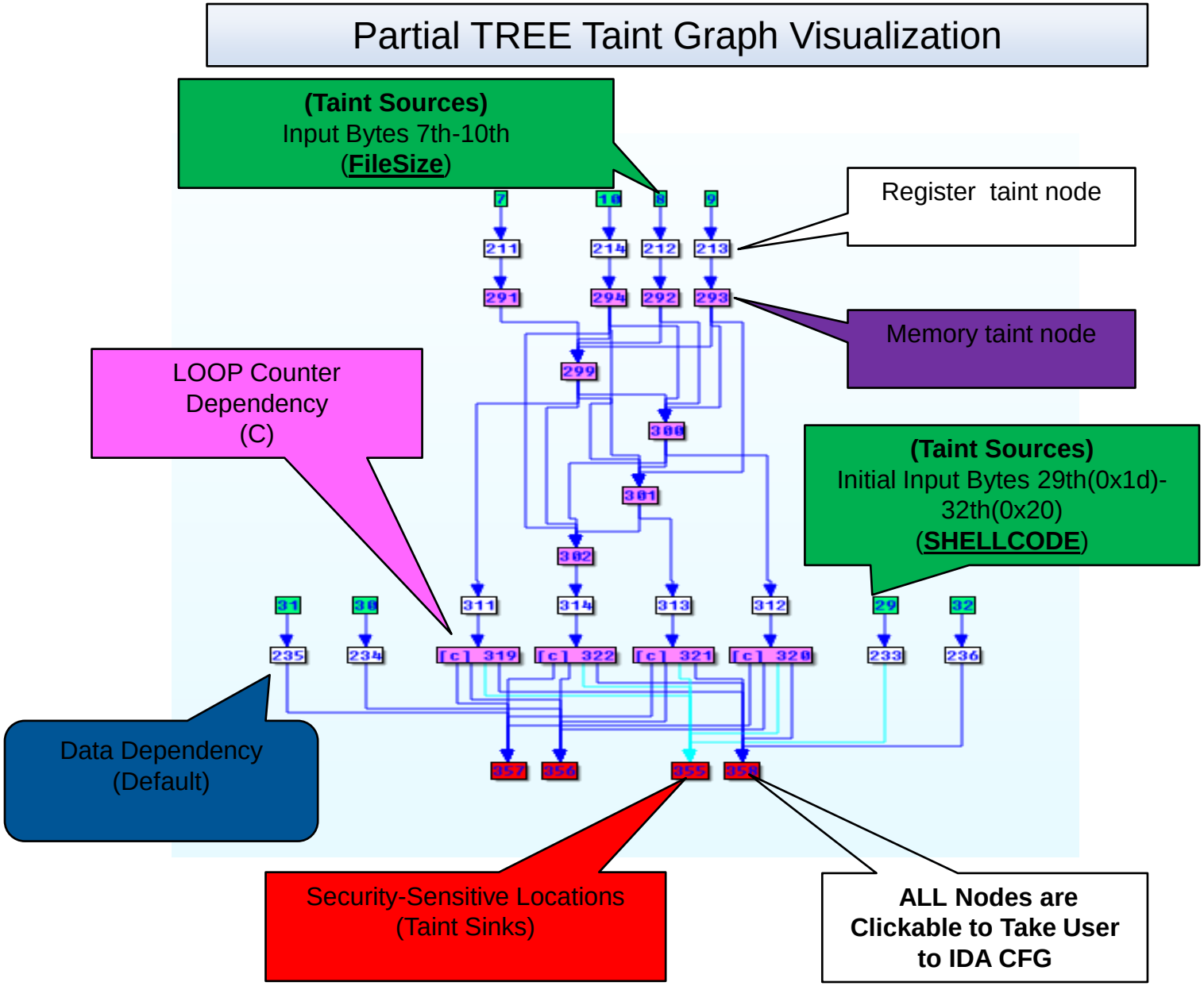


  - One type of record is “escape” record
  - SETABORTPROC escape allow an application to register a hook function to handle spooler errors

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

WMF Taint Graph

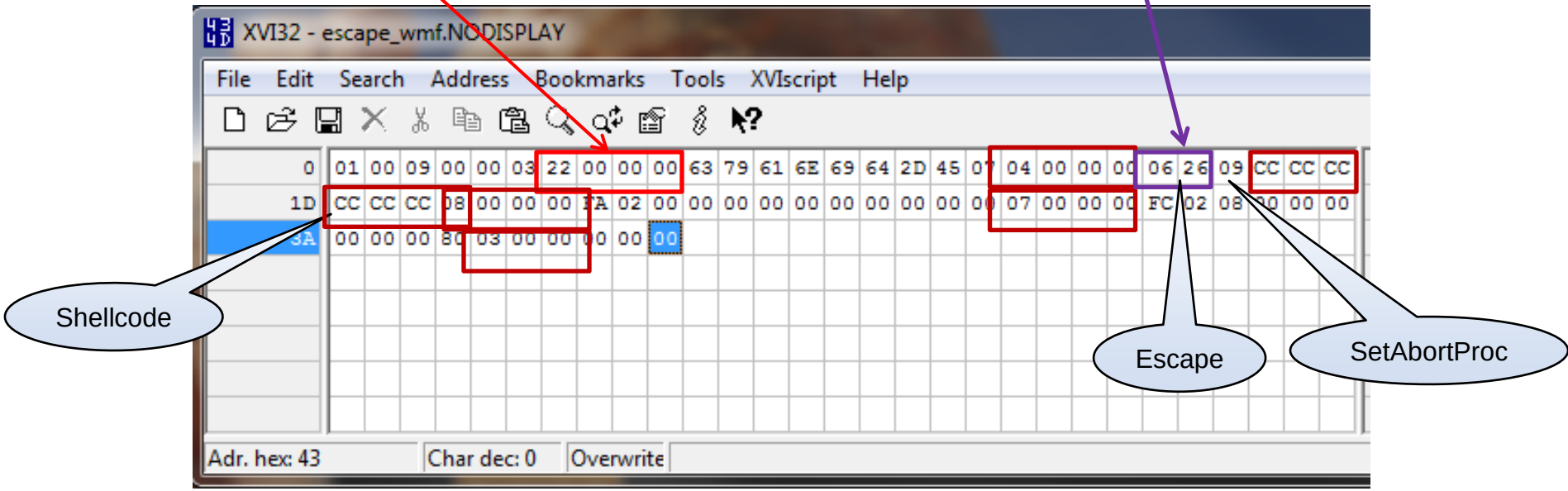


WMF File: The Fields & The Vulnerability

- 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;
```



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=0xa) R 4 a9f8c 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 a87ee 22_0_0_0

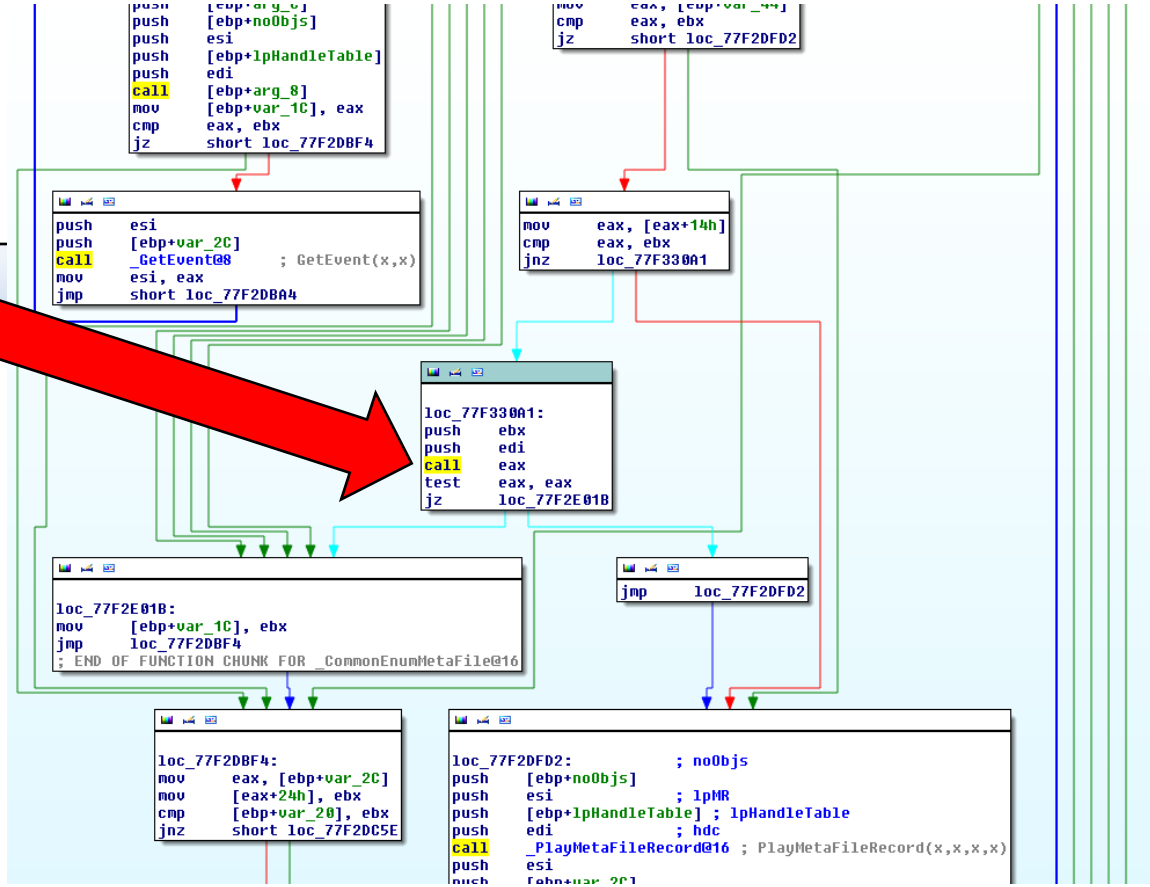
0x77c472e3 rep movsd 2 f3a5 0x0 0xb13c Reg(EDI=0xa87ec eflags=0x10216 ESI=0xa9f74 ECX=0x10) R 4 a9f74 0_3_22_0 W 4 a87ec

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

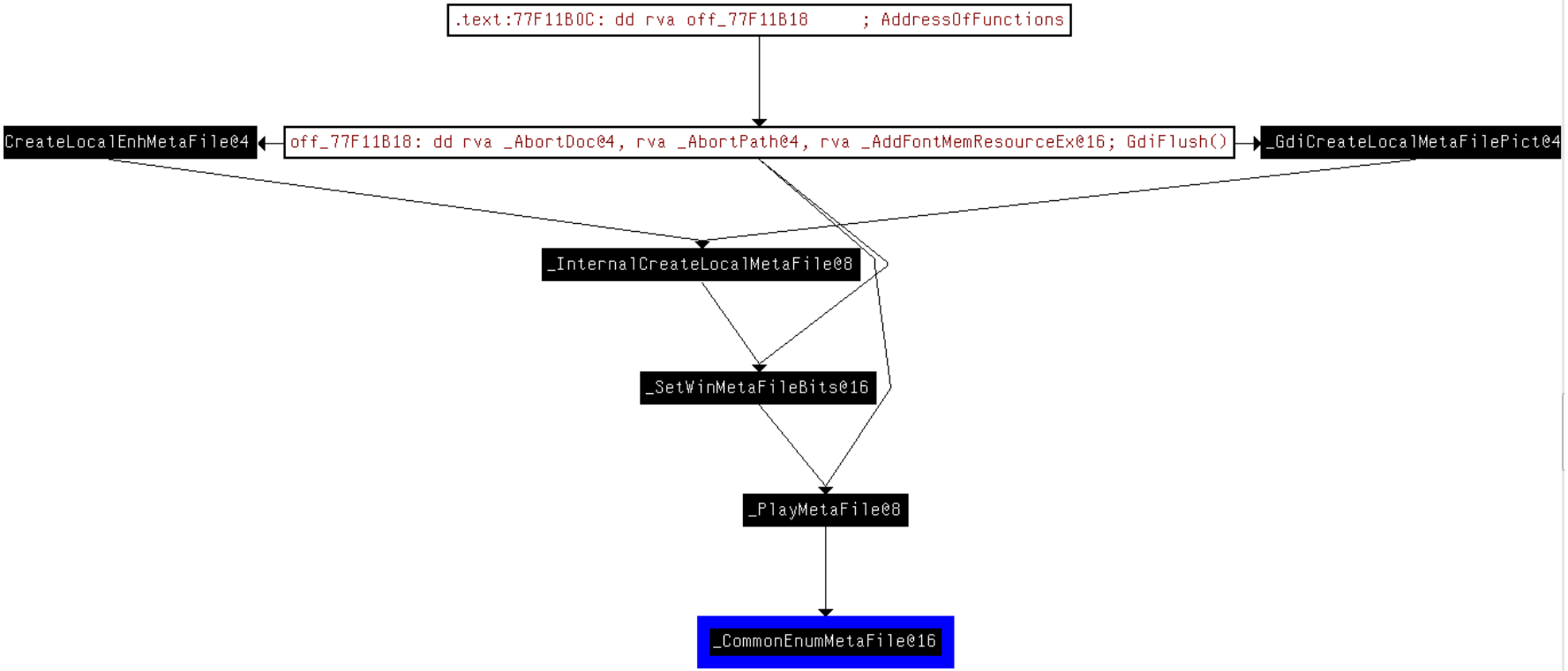


WMF Slicing (3)

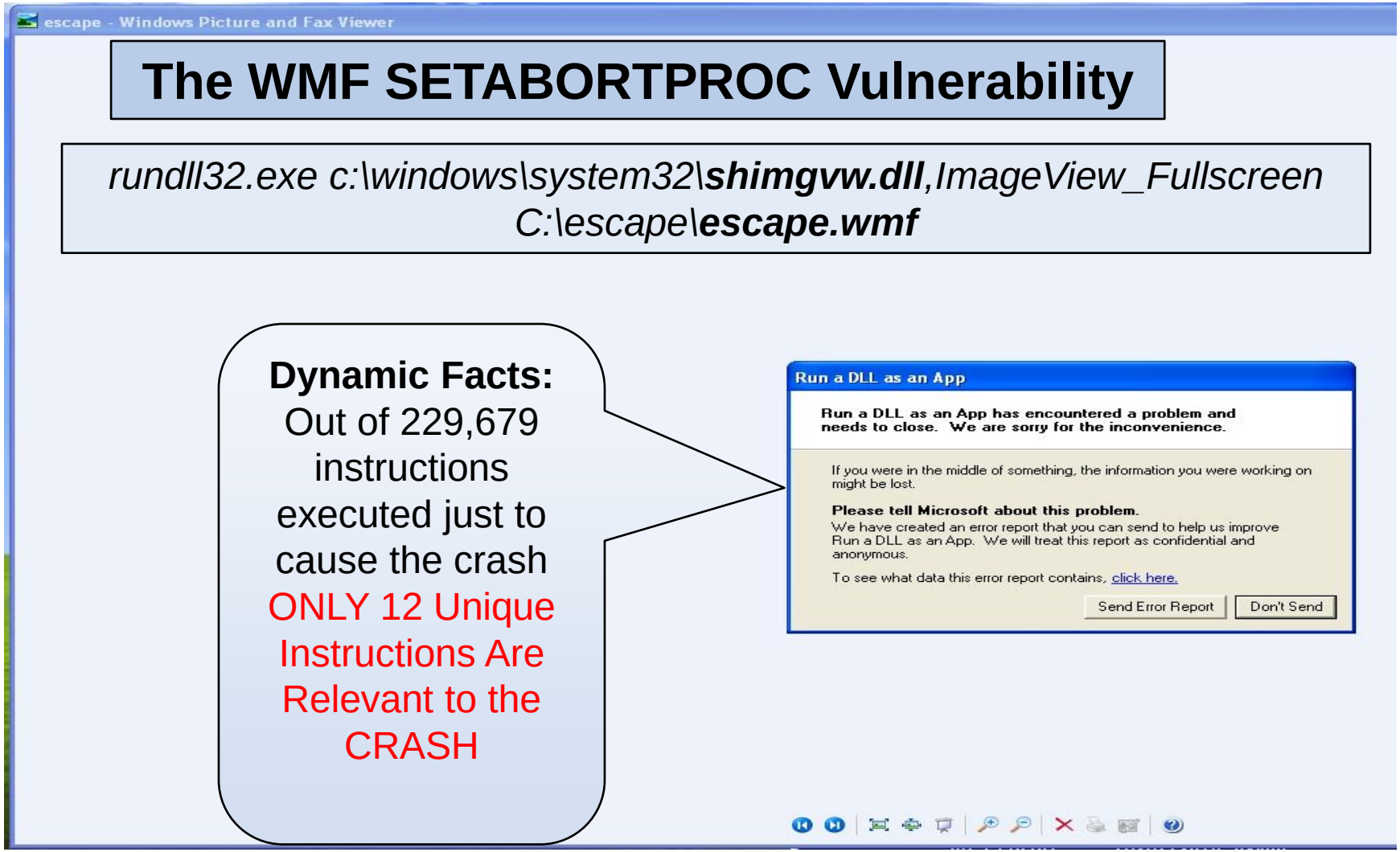
An Instruction Slice with Text Helps a Little

text:77F330A3 call eax

More Context Helps More
Module: gdi32.dll
Function: CommonEnumMetaFile
Call Graph: caller *PlayMetaFile*



WMF -- The Relevant Parts



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, Project Lead
  - xingzli@gmail.com, Developer
  - locvnguy@gmail.com, Developer
  - 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