Taint Nobody Got Time for Crash Analysis

Crash Analysis

Triage Goals

Execution Path

- What code paths were executed
- What parts of the execution interacted with external data

Input Determination

- Which input bytes influence the crash

Exploitability

- Does this crash have a security impact
  - Read Access – Information Leak
    - ASLR Bypass
  - Write Access – Data Modification
    - Credentials
    - Control Flow
  - Execute Access – Game Over

Common Scenarios

Fuzzing
- Spray ‘n Pray
- Grammar-based
- “Fuzzing with Code Fragments”

Static Analysis
- Intra-procedural Analysis Tools
- Manual code review

Third Party
- In-the-wild exploitation
- Vulnerability response teams
- Vulnerability brokers

Existing Tools

Execution Path

- Process Stalker, CoverIt (hexblog), BlockCov, IDA PIN Block Trace
- Bitblaze, Taintgrind, VDT

Input Determination

- delta, tmin, diff

Exploitability

- !exploitable
- CrashWrangler
- CERT Triage Tools

Automation Methods

Execution Path
- Code Coverage
- Taint Analysis

Input Determination
- Slicing

Exploitability
- Symbolic Execution
- Abstract Interpretation

Automation Methods

Execution Path

- Code Coverage
- **Taint Analysis**

Input Determination

- **Slicing**

Exploitability

- **Symbolic Execution**
- Abstract Interpretation

Taint Analysis

Concept

Formally – Information Flow Analysis

- Type of dataflow analysis
- Can be static or dynamic, often hybrid
- Applied to track user controlled data through execution

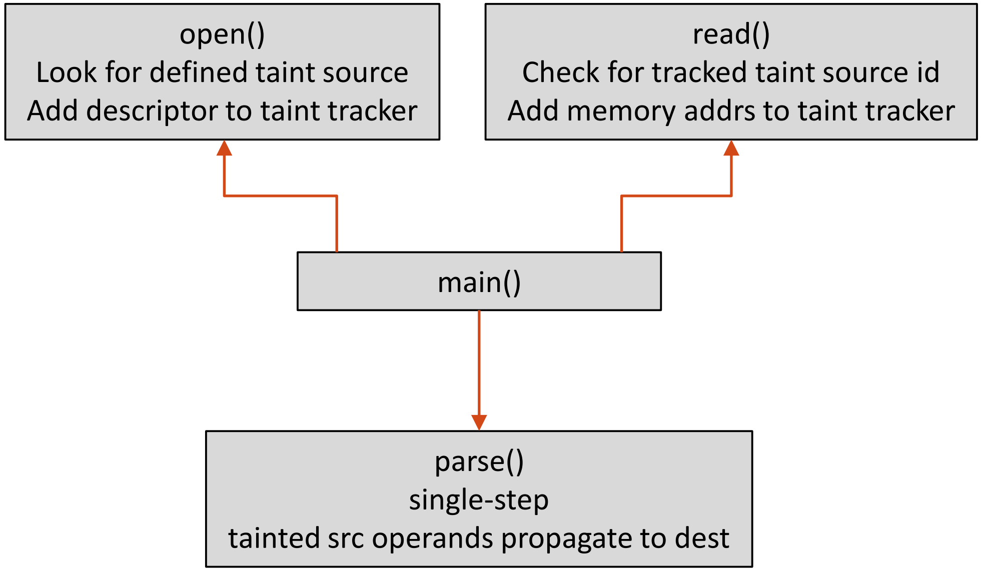
Methodology

- Define taint sources
- Single-step execution
- Apply taint propagation policy for each instruction
- Apply taint checks (if any)

Concept

Define Taint Sources

- Hook I/O Functions
- Look for taint sources
  - File name, network ip:port, etc
  - Track tainted file descriptor
- Single-step
- Add future data reads from taint source descriptors to the taint tracking engine
- Apply taint policy on each instruction



Concept

Define Taint Sources

- Hook I/O Functions
- Look for taint sources
  - File name, network ip:port, etc
  - Track tainted file descriptor
- Single-step
- Add future data reads from taint source descriptors to the taint tracking engine
- Apply taint policy on each instruction

Explicit Taint Propagation

```
A = TAINT()
B = A
C = B + 1
D = C * B
E = *(D)
```

Implicit Taint Propagation

```
A = TAINT()
IF A > B:
     C = TRUE
ELSE:
     C = FALSE
```

Implementation Details

We utilize a tracer forked from the Binary Analysis Platform from Carnegie-Mellon University to facilitate taint tracing

- Originally wrote separate PIN based tracer
- BAP’s tracer is also a Pintool
- Worked with the authors of BAP since early 2012 to improve the tracer so it performs acceptably against complex COTS software targets on Windows
- Added code coverage and memory dump collection to our private version

PIN supplies a robust API and framework for binary instrumentation

- Supports easily hooking I/O functions for taint sources
- High performance single-stepping
- Supports instrumenting at instruction level for taint propagation / checks

Implementation Details

Taint Propagation Policy

- Tree of tainted references to registers and bytes of memory are individually tracked
- If input operands contain taint, propagate to all output operands
- No control flow tainting
- Optionally taint index registers
  - All index registers for LEA instructions are tainted
- No support for MMX, Floating point FCMOV, SSE PREFETCH

Taint Visualization Demo

```
.text:08048871
.text:08048872
.text:08048872 ; =============== S U B R O U T I N E =======================================
.text:08048872
.text:08048872
.text:08048872 foo             proc near               ; CODE XREF: nice_crash↑p
.text:08048872
.text:08048872 arg_4           = dword ptr  8
.text:08048872
.text:08048872                 mov     esi, [esp+arg_4]
.text:08048876                 xor     eax, eax
.text:08048878                 lodsb                   ;
.text:08048878                                         ;   @context "R_EAX" = 0x0, 0, u32, wr @context "R_ESI" = 0x9cb0000, 0, u32, rd
.text:08048878                                         ;   @context "EFLAGS" = 0x246, 0, u32, rd
.text:08048878                                         ;   @context "mem[0x9cb0000]" = 0x41, 1, u8, rd
.text:08048878                                         ; label pc_0x8048878
.text:08048878                                         ; T_32t0:u32 = R_DFLAG:u32
.text:08048878                                         ; T_32t1:u32 = R_ESI:u32
.text:08048878                                         ; T_8t2:u8 = mem:?u32[T_32t1:u32, e_little]:u8
.text:08048878                                         ; R_EAX:u32 = R_EAX:u32 & 0xffffff00:u32 | pad:u32(T_8t2:u8)
.text:08048878                                         ; T_32t3:u32 = T_32t1:u32 + T_32t0:u32
.text:08048878                                         ; R_ESI:u32 = T_32t3:u32
.text:08048878                                         ;
.text:08048879                 xor     edi, edi
.text:0804887B                 add     edi, eax        ;
.text:0804887B                                         ;   @context "R_EDI" = 0x0, 0, u32, rw
.text:0804887B                                         ;   @context "R_EAX" = 0x41, 1, u32, rd @context "EFLAGS" = 0x246, 0, u32, wr
.text:0804887B                                         ; label pc_0x804887b
.text:0804887B                                         ; T_t1:u32 = R_EDI:u32
.text:0804887B                                         ; T_t2:u32 = R_EAX:u32
.text:0804887B                                         ; R_EDI:u32 = R_EDI:u32 + T_t2:u32
.text:0804887B                                         ; R_CF:bool = R_EDI:u32 < T_t1:u32
.text:0804887B                                         ; R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EDI:u32 ^ T_t1:u32 ^ T_t2:u32))
.text:0804887B                                         ; R_OF:bool = high:bool((T_t1:u32 ^ ~T_t2:u32) & (T_t1:u32 ^ R_EDI:u32))
.text:0804887B                                         ; R_PF:bool =
.text:0804887B                                         ;   ~low:bool(R_EDI:u32 >> 7:u32 ^ R_EDI:u32 >> 6:u32 ^ R_EDI:u32 >> 5:u32 ^
.text:0804887B                                         ;             R_EDI:u32 >> 4:u32 ^ R_EDI:u32 >> 3:u32 ^ R_EDI:u32 >> 2:u32 ^
.text:0804887B                                         ;             R_EDI:u32 >> 1:u32 ^ R_EDI:u32)
.text:0804887B                                         ; R_SF:bool = high:bool(R_EDI:u32)
.text:0804887B                                         ; R_ZF:bool = 0:u32 == R_EDI:u32
.text:0804887B                                         ;
.text:0804887D                 sub     edi, 30h        ;
.text:0804887D                                         ;   @context "R_EDI" = 0x41, 1, u32, rw
.text:0804887D                                         ;   @context "EFLAGS" = 0x206, 1, u32, wr
.text:0804887D                                         ; label pc_0x804887d
.text:0804887D                                         ; T_t:u32 = R_EDI:u32
.text:0804887D                                         ; R_EDI:u32 = R_EDI:u32 - 0x30:u32
.text:0804887D                                         ; R_CF:bool = T_t:u32 < 0x30:u32
.text:0804887D                                         ; R_OF:bool = high:bool((T_t:u32 ^ 0x30:u32) & (T_t:u32 ^ R_EDI:u32))
.text:0804887D                                         ; R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EDI:u32 ^ T_t:u32 ^ 0x30:u32))
.text:0804887D                                         ; R_PF:bool =
.text:0804887D                                         ;   ~low:bool(R_EDI:u32 >> 7:u32 ^ R_EDI:u32 >> 6:u32 ^ R_EDI:u32 >> 5:u32 ^
.text:0804887D                                         ;             R_EDI:u32 >> 4:u32 ^ R_EDI:u32 >> 3:u32 ^ R_EDI:u32 >> 2:u32 ^
.text:0804887D                                         ;             R_EDI:u32 >> 1:u32 ^ R_EDI:u32)
.text:0804887D                                         ; R_SF:bool = high:bool(R_EDI:u32)
.text:0804887D                                         ; R_ZF:bool = 0:u32 == R_EDI:u32
.text:0804887D                                         ;
```

Design Considerations

Taint Policy

- Implicit Information Flows
  - Over-tainting
    - Most common when applying implicit taint via control flow
  - Under-tainting
    - If control flow taint is ignored

Performance

- Execution Speed
  - Analysis on each instruction is expensive
  - Avoid context switching
- Memory Overhead

Trace Slicing

Concept

Trace slicing finds the sub-graph of dependencies between two nodes

- All nodes that influence or are influenced by specified node can be isolated
- Reachability Problem

Forward Slicing

- Slice forward to determine instructions influenced by selected value

Backward Slicing

- Slice backward to locate the instructions influencing a value
- Collect constraints to determine the degree of control over the value

Concept

Methodology

- Collect trace
- Convert native assembler to IL
- Select location and value of interest (register or memory address)
- Select direction of slice
- Follow dependencies in desired direction to produce sub-graph

Forward Slicing

Slice forward to determine instructions influenced by a value

```
S = {v}
For each stmt in statements:
    If vars(stmt.rhs) ∩ S != ∅ then
        S := S ∪ {stmt.lhs}
    else
        S := S – {stmt.lhs}
Return S
```

stmt	S
el_size, el_count, el_data = read()	{**el_size**}
total_size = **el_size** * el_count	{**el_size**, **total_size**}
buf = malloc(**total_size**)	{el_size, **total_size**}
while count < el_count	{el_size, total_size}
offset = count * **el_size**	{**el_size**, total_size, **offset**}
data_offset = el_data + **offset**	{el_size, total_size, **offset**, **data_offset**}
buf_offset = buf + **offset**	{el_size, total_size, **offset**, data_offset, **buf_offset**}
memcpy(buf_offset, **data_offset**, **el_size**)	{**el_size**, total_size, offset, data_offset, **buf_offset**}

Backward Slicing

Slice backward to locate the instructions influencing a value

```
S = {v}
For each stmt in reverse(statements):
    If {stmt.lhs} ∩ S != ∅ then
        S := S – {stmt.rhs}
        S := S ∪ vars(stmt.rhs)
Return S
```

stmt	S
el_size, el_count, **el_data** = read()	{data_offset, **el_data**, offset, count, el_size}
total_size = el_size * el_count	{data_offset, el_data, offset, count, el_size}
buf = malloc(total_size)	{data_offset, el_data, offset, count, el_size}
while count < el_count	{data_offset, el_data, offset, count, el_size}
offset = **count** * **el_size**	{data_offset, el_data, **offset, count, el_size**}
data_offset = **el_data** + **offset**	{**data_offset**, **el_data**, **offset**}
buf_offset = buf + offset	{data_offset}
memcpy(buf_offset, **data_offset**, el_size)	{**data_offset**}

Implementation Details

BAP includes an intermediate assembly language definition called BIL

BIL expands each native assembly instruction into a sequence of micro operations that make native instruction side effects explicit

We only have to handle assignments of the form *var := exp*

We concretize the trace and convert to SSA to create uniqe labels for each assignment

program	::=	*stmt**
stmt	::=	*var* := *exp* | `jmp`(*exp*) | `cjmp`(*exp*,*exp*,*exp*)
		| `halt`(*exp*) | `assert`(*exp*) | `label` *label_kind*
		| `special`(string)

Implementation Details

BAP includes an intermediate assembly language definition called BIL

BIL expands each native assembly instruction into a sequence of micro operations that make native instruction side effects explicit

We only have to handle assignments of the form *var := exp*

We concretize the trace and convert to SSA to create uniqe labels for each assignment

```
.text:08048887  mov edx, [edi+11223344h] ;
.text:08048887    ;   @context "R_EDX" = 0x1000, 0, u32, wr
.text:08048887    ;   @context "R_EDI" = 0x11, 1, u32, rd
.text:08048887    ;   @context "mem[0x11223355]" = 0x0, 0, u8, rd
.text:08048887    ;   @context "mem[0x11223356]" = 0x0, 0, u8, rd
.text:08048887    ;   @context "mem[0x11223357]" = 0x0, 0, u8, rd
.text:08048887    ;   @context "mem[0x11223358]" = 0x0, 0, u8, rd
.text:08048887    ; label pc_0x8048887
.text:08048887    ; R_EDX:u32 = mem:?u32[R_EDI:u32 + 0x11223344:u32, e_little]:u32
```

Backslice Demo

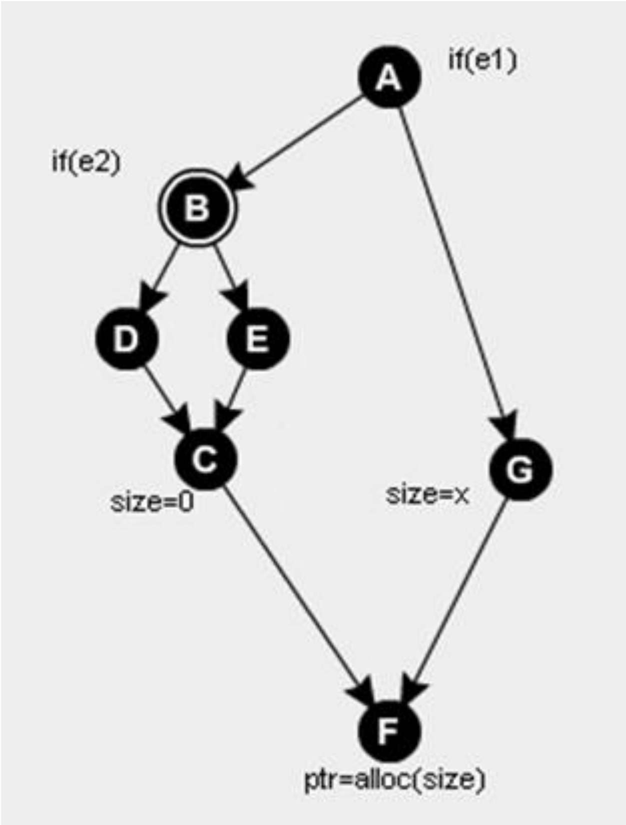
Design Considerations

Under-tainting Implicit Flows

- Backslice by “size” stops at node C because of a constant assignment
  - “size” is implicitly dependent on e1, but not on e2

Over-tainting

- APIs that hold state created by a previously tainted value may indicate taint in later calls
- Inflates the trace size by including calls with untainted arguments
- Example: malloc(tainted_size) could permanently taint the allocator’s internal structures



Symbolic Execution

Concept

Symbolic execution lets us "execute" a series of instructions without using concrete values for variables

Instead of a numeric output, we get a formula for the output in terms of input variables that represents a potential range of values

Given a crash state, analyze potential paths to find exploitable condition

- A path is exploitable if it meets prior path constraints and contains a tainted memory write or control transfer

Concept

Methodology

- Pick an initial state
  - Trace taint until point of interest
  - Store process state and memory image
- Choose desired future state
  - Depth-First Search for all future states
- Encode program logic from initial state to future state into SMT formula
- Initialize values in the SMT formula with saved program state
  - Replace one or more concrete values with symbolic value
- Solve formula with SMT solver

SMT Solvers

In very simple terms

- You ask a question, solver tries to answer

Question:

```
work, sleep, lulz = Ints('work sleep lulz')

solve(work >= 40,    # 40+ hour work week
      sleep >= 42,   # 6+ hours sleep/day
      lulz >= work,  # work/lulz balance
      work + sleep + lulz == 168) # 168 hours/week
```

Answer:

```
[sleep = 42, lulz = 63, work = 63]
```

SMT Solvers

In very simple terms

- You ask a question, solver tries to answer

Question:

```
x, y = Reals('x y')

solve(x**2 + y**2 < 1,
      2*x + y > 1),
      z**2 == 1/(x * y))
```

Answer:

```
[x = 1/8, y = 7/8, z = -3.0237157840?]
```

SMT Solvers

What’s the point?

- Translate program’s code into SMT-acceptable format
- Ask questions and possibly get some answers!

```
add eax, ebx
xor ebx, ebx
sub ecx, 0x123
setz bl
add eax, ebx
```

Is this snippet equivalent to “*`add eax, ebx`*”?

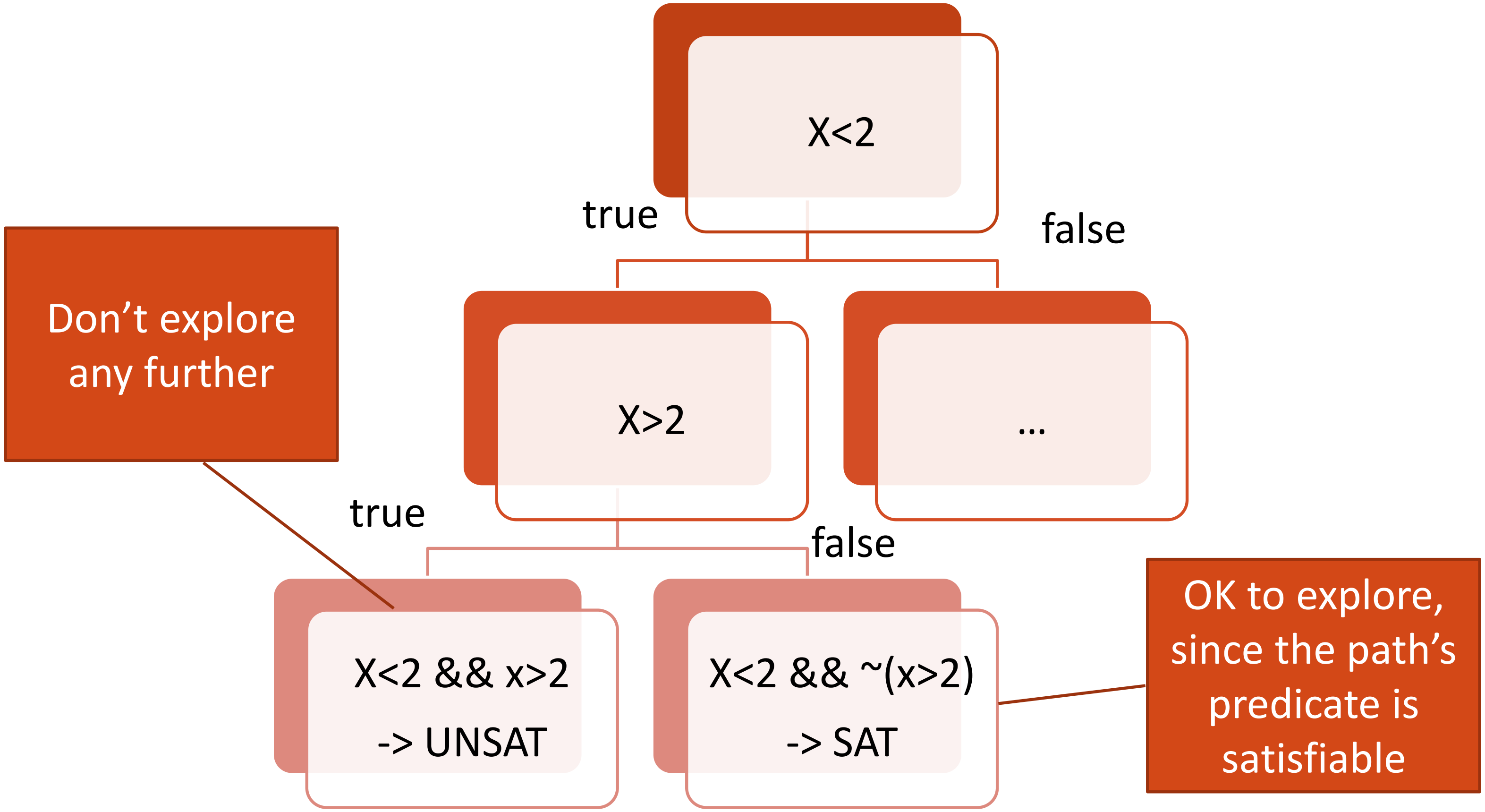
SMT Solvers

```
add eax, ebx
xor ebx, ebx
sub ecx, 0x123
setz bl
add eax, ebx
```

```
ASSERT( 0bin1 = (LET initial_EBX_77_0 = R_EBX_6 IN
(LET initial_EAX_78_1 = R_EAX_5 IN
(LET R_EAX_80_2 = BVPLUS(32, R_EAX_5,R_EBX_6) IN
(LET R_ECX_117_3 = BVSUB(32, R_ECX_7,0hex00000123) IN
(LET R_ZF_144_4 = IF (0hex00000000=R_ECX_117_3) THEN
0bin1 ELSE 0bin0 ENDIF IN
(LET R_EAX_149_5 = BVPLUS(32, R_EAX_80_2,
(0bin0000000000000000000000000000000 @ R_ZF_144_4)) IN
(LET final_EAX_180_6 = R_EAX_149_5 IN
IF (NOT(final_EAX_180_6=BVPLUS(32,
initial_EAX_78_1,initial_EBX_77_0))) THEN
);
QUERY(FALSE);
COUNTEREXAMPLE;
```

```
Model:
R_ECX_7 -> 0x123
Solve result: Invalid
```

Satisfiability



Implementation details

BAP’s tracer has been modified to collect registers, taint information and a memory snapshot when a crash occurs

Symbolic executor (motriage) uses this state as a starting point

motriage continues execution using variables instead of constants for unmapped memory:

```
mov eax, [ebx] => eax := new_variable() iff [ebx] is undefined
```

Taint is propagated for each instruction

Each instruction’s semantics is appended to our formula, using symbolic variables where necessary

Implementation details

For each code branch, motriage forks its state (registers, memory, taint info) and updates the current path's predicate:

- True path: $path\_pred \wedge cond$
- False path: $path\_pred \wedge \sim cond$

The SMT formula is then solved for each new path

- If the path's predicate becomes UNSAT, stop exploring that path

Continue the DFS search until SUCCESS or FAIL condition is met

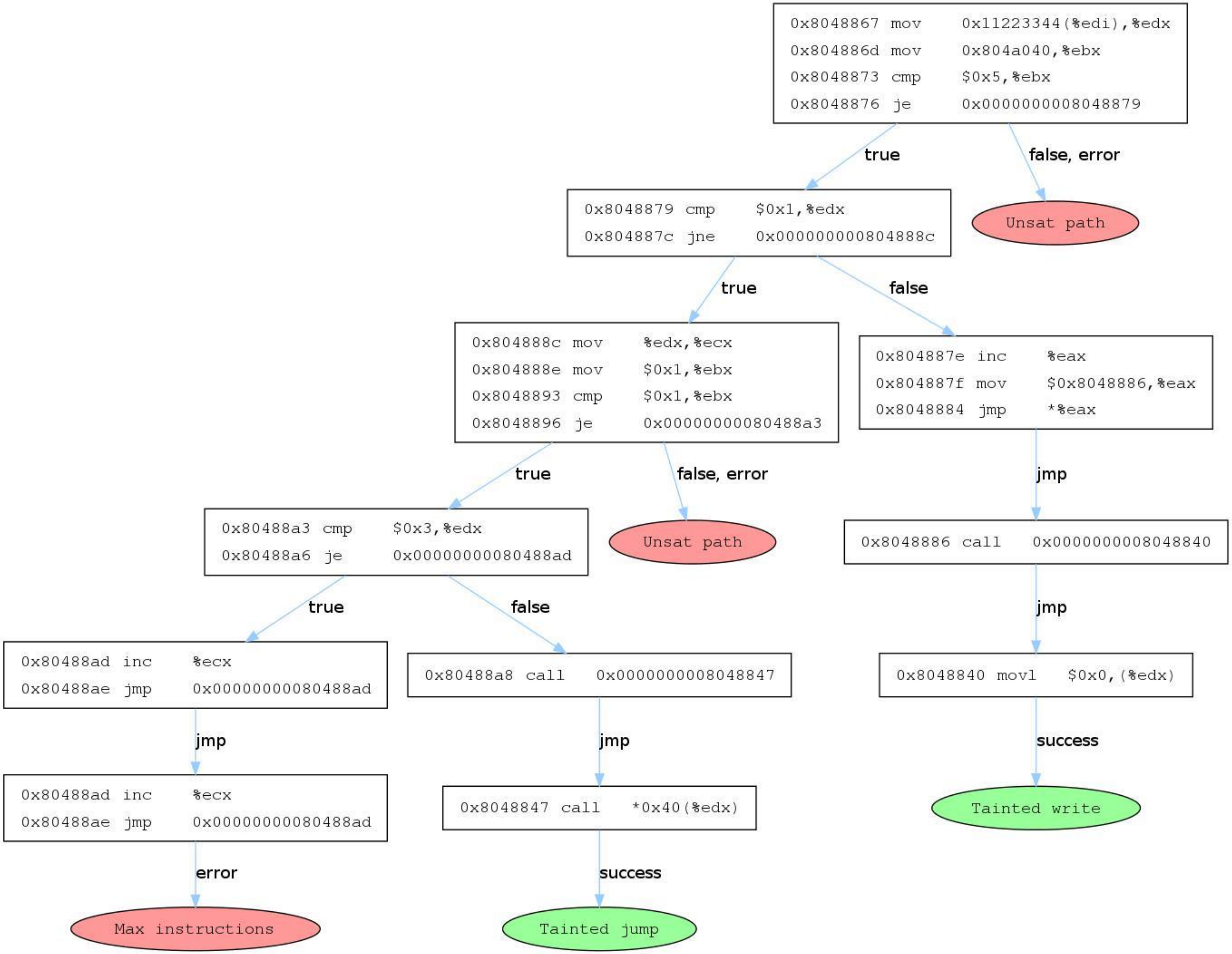
Implementation Details

Terminate with FAIL condition, if:

- Path is unsatisfiable (determined with a SMT solver):

```
X=1
If(x==2){ A }
Else { B }
```

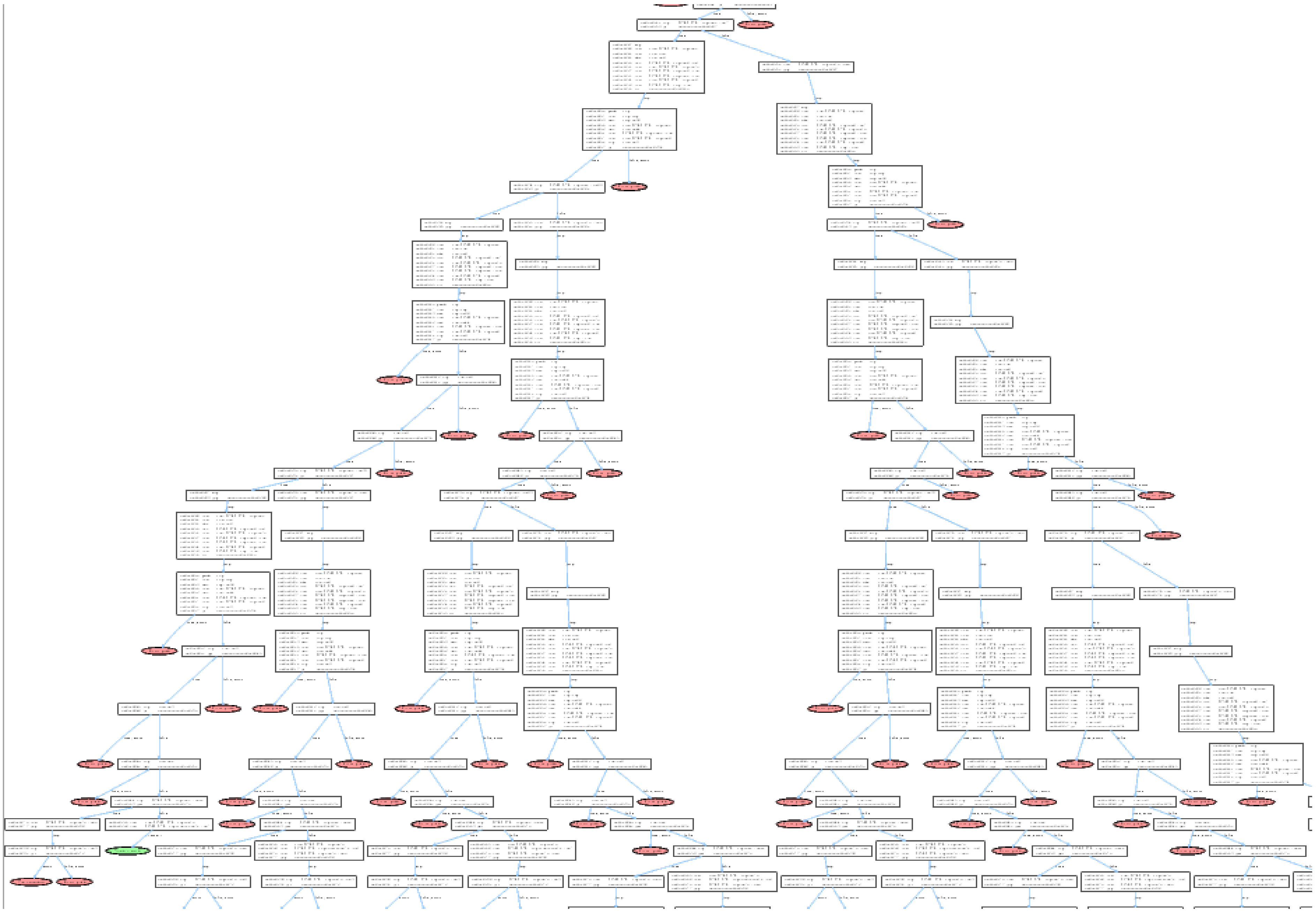
  - "A" can't be reached, so it's not analyzed
- Unknown (and untainted) jump target
  - We can't follow `jmp eax`, if eax is symbolic
- Symbolic (and untainted) write
  - `mov [eax], 0`
  - If EAX is symbolic but untainted, then we have no idea where exactly are we writing
  - All future reads would have to take that into account – too much trouble
- Max number of instructions or branches

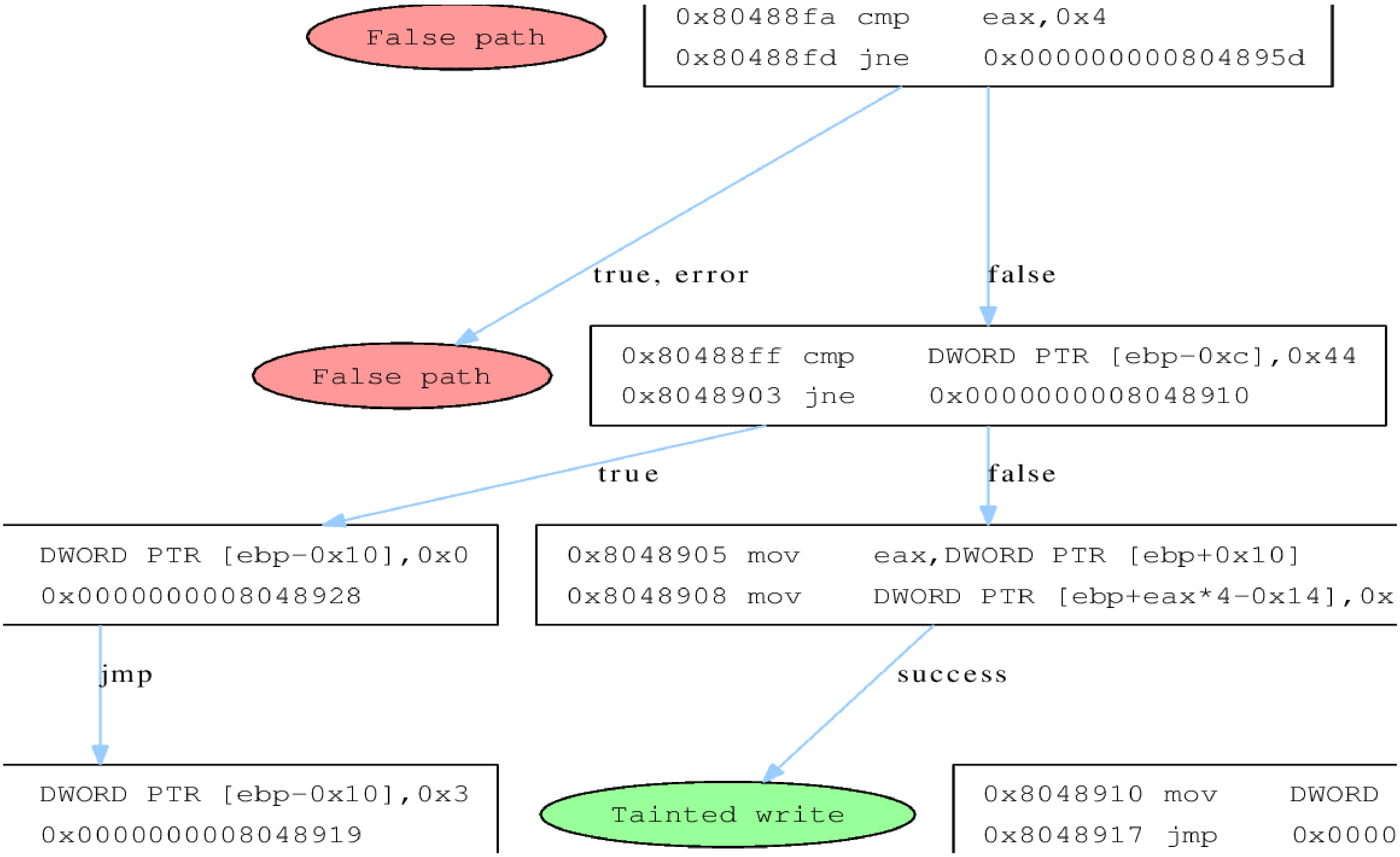

0x8048867 mov 0x11223344(%edi),%edx
0x804886d mov 0x804a040,%ebx
0x8048873 cmp $0x5,%ebx
0x8048876 je 0x0000000008048879
true
false, error
0x8048879 cmp $0x1,%edx
0x804887c jne 0x000000000804888c
Unsat path
true
false
0x804888c mov %edx,%ecx
0x804888e mov $0x1,%ebx
0x8048893 cmp $0x1,%ebx
0x8048896 je 0x00000000080488a3
0x804887e inc %eax
0x804887f mov $0x8048886,%eax
0x8048884 jmp *%eax
true
false, error
jmp
0x80488a3 cmp $0x3,%edx
0x80488a6 je 0x00000000080488ad
Unsat path
0x8048886 call 0x0000000008048840
true
false
jmp
0x80488ad inc %ecx
0x80488ae jmp 0x00000000080488ad
0x80488a8 call 0x0000000008048847
0x8048840 movl $0x0,(%edx)
jmp
jmp
success
0x80488ad inc %ecx
0x80488ae jmp 0x00000000080488ad
0x8048847 call *0x40(%edx)
Tainted write
error
success
Max instructions
Tainted jump

Triage Tool Demo

```c
void test_motriage(unsigned int
*buf)
{
  unsigned int b,x,y;
  b = buf[0];
  x = buf[b+0x11223344];
  y = buf[x];
  exploit_me(1, x, y);
}
```

```c
void exploit_me(int depth, unsigned int
x, unsigned int y)
{
  int stack[1];
  int b, i;
  b = x & 0xff;
  switch(depth){
    case 4:
      if(b == 0x44)
        stack[y] = 1;
      for(i=0; i < 4; i++)
        stack[i] = 0x29a;
      return;
    case 3:
      if(b != 0x33) y = 0;
      break;
    case 2:
      if(b != 0x22) y = 0;
      break;
    case 1:
      if(b != 0x11) y = 0;
      break;
    default:
      assert(0);
  }
  exploit_me(++depth, x>>8, y);
}
```





False path
0x80488fa cmp eax,0x4
0x80488fd jne 0x000000000804895d
true, error
false
False path
0x80488ff cmp DWORD PTR [ebp-0xc],0x44
0x8048903 jne 0x0000000008048910
true
false
DWORD PTR [ebp-0x10],0x0
0x0000000008048928
0x8048905 mov eax,DWORD PTR [ebp+0x10]
0x8048908 mov DWORD PTR [ebp+eax*4-0x14],0x
jmp
success
DWORD PTR [ebp-0x10],0x3
0x0000000008048919
Tainted write
0x8048910 mov DWORD
0x8048917 jmp 0x0000

Performance

Two factors: number of branches and code size

- Running time exponential in number of branches
- N branches require n forks, so 2^n possible paths to analyze
- For branchless code you pay the same as in a software emulator (linear time)

How deep do you want to search?

- First, you need to get to the controlled write without crashing
- Then you need to perform a write to address constrained by all the conditional branches you passed
- The farther the write is, the less likely it’s going to be useful
- Eventually path explosion will meet hardware limits

False positives

False Positives

- Every read from unmapped (or symbolic) address creates a new symbolic variable
- We don’t know what exactly we are reading, so we don’t know what constraints should be asserted on these variables
- Consider an example:

Let x,y be tainted variables and for all i, mem[i] % 2 == 0

```
z = mem[x];
if(z % 2 == 1) {
   mem[y] = 0;
}
```

- Our approach incorrectly reports a SUCCESS on mem[y] = 0, despite this path being unsatisfiable

Conclusion

Value of a crash is related to our ability to perform difficult analysis

Automation solutions are needed to keep up with crash generation

Combined with slicing, taint analysis greatly reduces manual analysis time for gathering data flow information

Symbolic execution, seeded with taint information, allows us automatically to reason about exploitability of a crash with a higher degree of accuracy that previous solutions

Thank You

Richard Johnson
@richinseattle
moflow.org

pa_kt
@pa_kt
gdtr.wordpress.com

A special thanks to Ed Schwartz and the rest of the team working on BAP

- http://bap.ece.cmu.edu/