Dynamic Binary Analysis and Instrumentation Covering a function using a DSE approach

Jonathan Salwan
jsalwan@quarkslab.com

Security Day
Lille – France

January 16, 2015

Keywords : Program analysis, DBI, Pin, concrete execution, symbolic execution, DSE, taint analysis, context snapshot and Z3 theorem prover.

Who am I?

- I am a junior security researcher at Quarkslab
- I have a strong interest in all low level computing and program analysis
- I like to play with weird things even though it's useless

roadmap of this talk

- Introduction
  - Dynamic binary instrumentation
  - Data flow analysis
  - Symbolic execution
  - Theorem prover
- Code coverage using a DSE approach
  - Objective
  - Snapshot
  - Registers and memory references
  - Rebuild the trace with the backward analysis
- DSE example
- Demo
- Some words about vulnerability finding
- Conclusion

Introduction

Introduction
Dynamic Binary Instrumentation

Dynamic Binary Instrumentation

- A DBI is a way to execute an external code before or/and after each instruction/routine

- With a DBI you can:
  - Analyze the binary execution step-by-step
    - Context memory
    - Context registers
  - <u>Only analyze the executed code</u>

Dynamic Binary Instrumentation

- How does it work in a nutshell?

```
initial_instruction_1
initial_instruction_2
initial_instruction_3
initial_instruction_4
```

```
jmp_call_back_before
initial_instruction_1
jmp_call_back_after

jmp_call_back_before
initial_instruction_2
jmp_call_back_after

jmp_call_back_before
initial_instruction_3
jmp_call_back_after

jmp_call_back_before
initial_instruction_4
jmp_call_back_after
```

Pin

- Developed by Intel
  - Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures
  - The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications in Linux and Windows

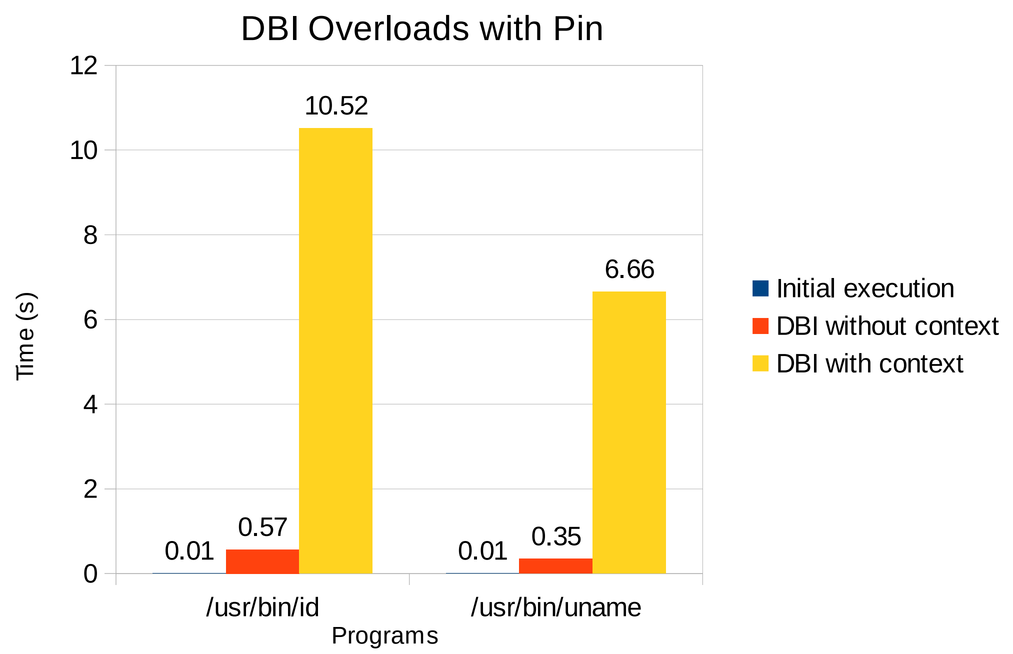
Pin tool - example

- Example of a provided tool: ManualExamples/inscount1.cpp
  - Count the number of instructions executed

```
VOID docount(UINT32 c) { icount += c; }

VOID Trace(TRACE trace, VOID *v) {

    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl)){
        BBL_InsertCall(bbl,
                       IPOINT_BEFORE,
                       (AFUNPTR)docount,
                       IARG_UINT32,
                       BBL_NumIns(bbl),
                       IARG_END);
}

int main(int argc, char * argv[]) {
     ...
     TRACE_AddInstrumentFunction(Trace, 0);
}
```

Dynamic Binary Instrumentation

- Dynamic binary instrumentation overloads the initial execution
  - The overload is even more if we send the context in our callback

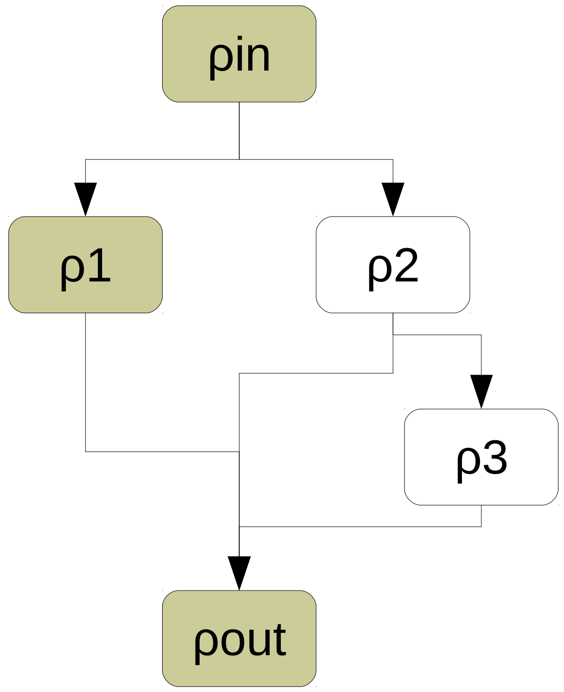


Dynamic Binary Instrumentation

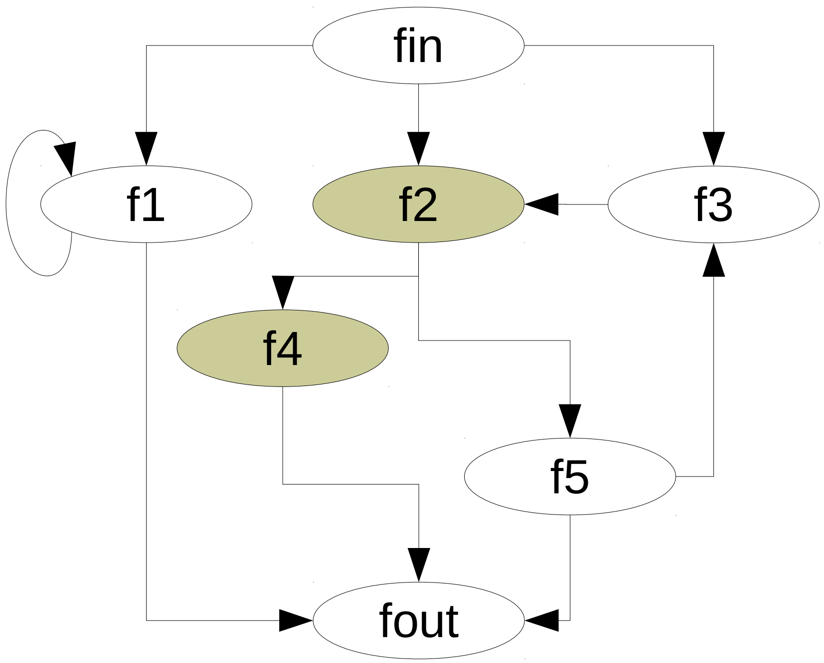
- Instrumenting a binary in its totality is unpractical due to the overloads
  - That's why we need to target our instrumentation
    - On a specific area
    - On a specific function and its subroutines
  - Don't instrument something that you don't want
    - Ex: A routine in a library
      - strlen, strcpy, …
    - We already know these semantics and can predict the return value with the input value

Dynamic Binary Instrumentation

- Target the areas which need to be instrumented



Control flow graph

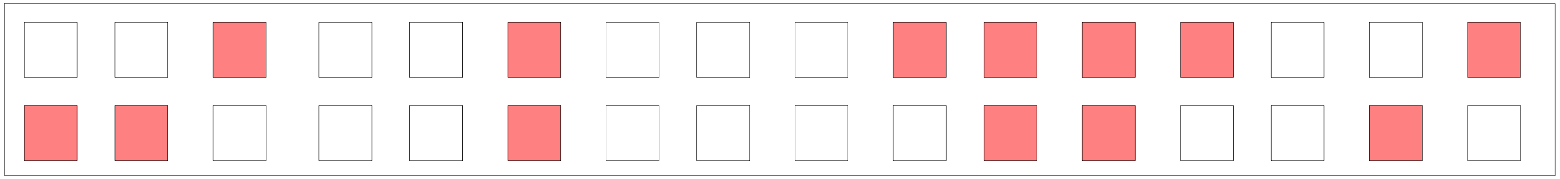


Call graph

Introduction
Data Flow Analysis

Data Flow Analysis

- Gather information about the possible set of values calculated at various points
- Follow the spread of variables through the execution
- There are several kinds of data flow analysis:
  - Liveness analysis
  - Range analysis
  - Taint analysis
    - Determine which bytes in memory can be controlled by the user ( )



Memory

Pin and Data Flow Analysis

- Define areas which need to be tagged as controllable
  - For us, this is the environment

```
int main(int argc, const char *argv[], const char *env[]) {...}
```

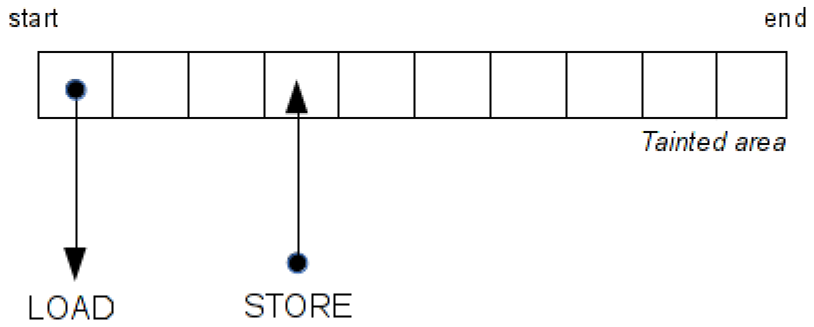
  - And syscalls

```
read(fd, buffer, size)
```

Example with sys_read() → For all “byte” in [buffer, buffer+size-1] (Taint(byte))

Pin and Data Flow Analysis

- Then, spread the taint by monitoring all instructions which read (*LOAD*) or write (*STORE*) in the tainted area




```
if (INS_MemoryOperandIsRead(ins, 0) &&
    INS_OperandIsReg(ins, 0)){

  INS_InsertCall(ins, IPOINT_BEFORE,
    (AFUNPTR)ReadMem,
    IARG_MEMORYOP_EA, 0,
    IARG_UINT32,INS_MemoryReadSize(ins),
    IARG_END);
}
```

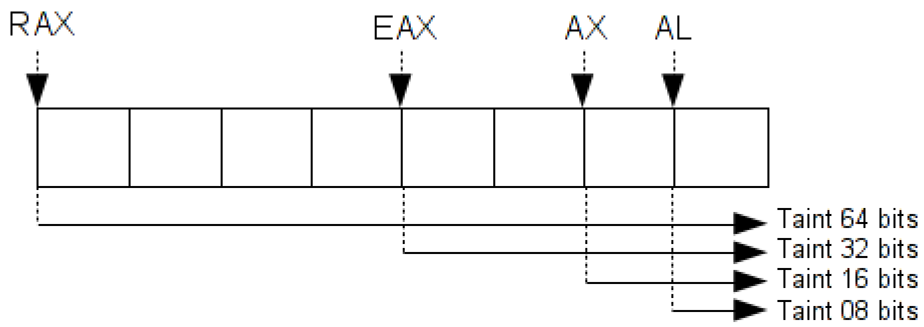
mov regA, [regB]

```
if (INS_MemoryOperandIsWritten(ins, 0)){

  INS_InsertCall(
    ins, IPOINT_BEFORE,(
    (AFUNPTR)WriteMem,
    IARG_MEMORYOP_EA, 0,
    IARG_UINT32,INS_MemoryWriteSize(ins),
    IARG_END);
}
```

mov [regA], regB.

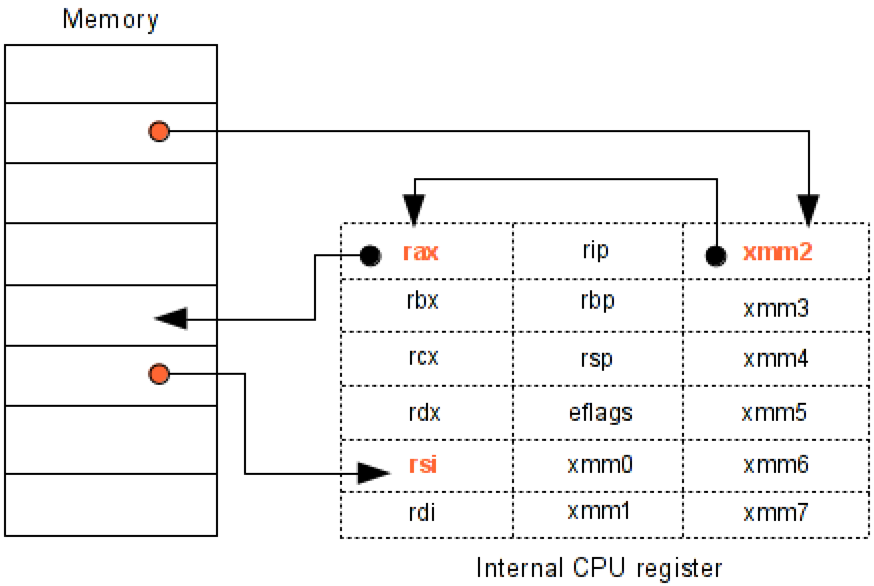
Pin and Data Flow Analysis

- Tainting the memory areas is not enough, we must also taint the registers.
  - More accuracy by tainting the bits
    - Increases the analysis's time



Pin and Data Flow Analysis

- So, by monitoring all STORE/LOAD and GET/PUT instructions, we know at every program points, which registers or memory areas are controlled by the user



Introduction
Symbolic Execution

Symbolic Execution

- Symbolic execution is the execution of the program using <u>symbolic variables instead of concrete values</u>
- Symbolic execution translates the program's semantic into a logical formula
- Symbolic execution can build and keep a path formula
  - By solving the formula, we can take all paths and “cover” a code
    - Instead of concrete execution which takes only one path
- Then a symbolic expression is given to a theorem prover to generate a concrete value

Symbolic Execution

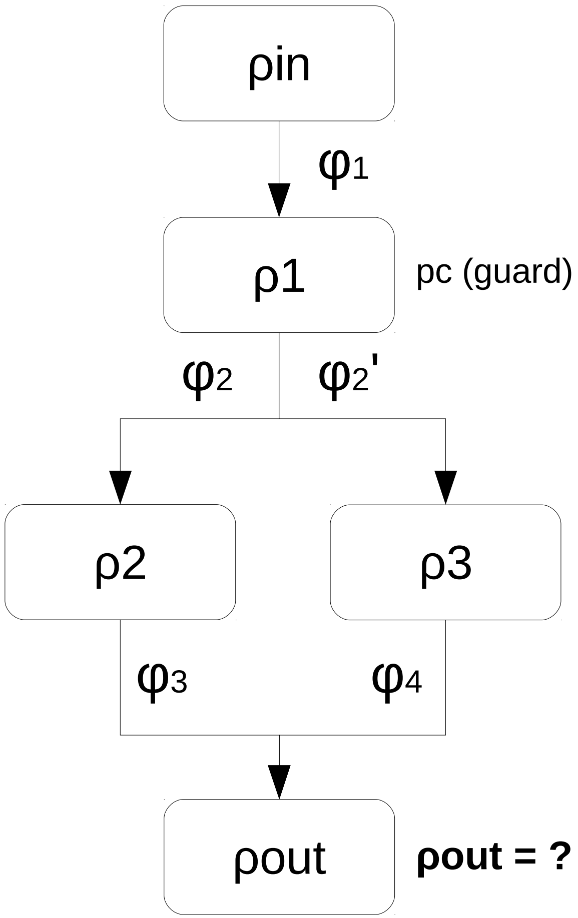
- There exists two kinds of symbolic execution
  - Static Symbolic Execution (SSE)
    - Translates program statements into formulae
      - Mainly used to check if a statement represents the desired property
  - Dynamic Symbolic Execution (DSE)
    - Builds the formula at runtime step-by-step
      - Mainly used to know how a branch can be taken
      - Analyze only one path at a time

Symbolic Execution

- Path formula
  - This control flow graph can take 2 different paths
    - What is the path formula for the ρout node?

$$\textbf{ρout} = \varphi_1 \wedge ((pc \wedge \varphi_2 \wedge \varphi_3) \vee (\neg pc \wedge \varphi_2' \wedge \varphi_4))$$

φ = statement / operation
pc = path condition (guard)



Control flow graph

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: {True} [x1 = i1]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: {True} [x1 = i1, y1 = i2]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 > 80 ?} [x1 = i1, y1 = i2]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 > 80} [x2 = y1 * 2, y1 = i2]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 > 80} [x2 = y1 * 2, y2 = 0]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 > 80 ∧ x2 == 256 ?} [x2 = y1 * 2, y2 = 0]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 > 80 ∧ x2 == 256} [x2 = y1 * 2, y2 = 0]
At this point φk can be taken iff (x1 > 80) ∧ (x2 == 256)

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 <= 80} [x1 = i1, y1 = i2]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 <= 80} [x2 = 0, y1 = i2]

Symbolic Execution

```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { x1 <= 80} [x2 = 0, y2 = 0]

Symbolic Execution

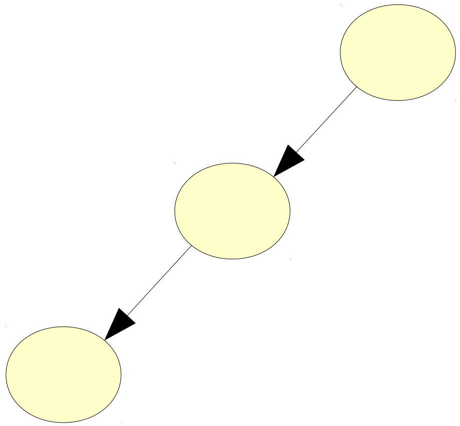
```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```

- SSE path formula and statement property

PC: { (x1 <= 80) ∨ ((x1 > 80) ∧ (x2 != 256)) }
[(x2 = 0, y2 = 0) ∨ (x2 = y1 * 2, y2 = 0)]

Symbolic Execution

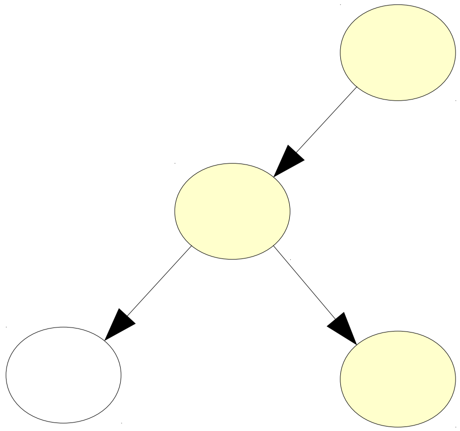
- With the DSE approach, we can only go through one single path at a time.



Paths discovered at the 1st iteration

Symbolic Execution

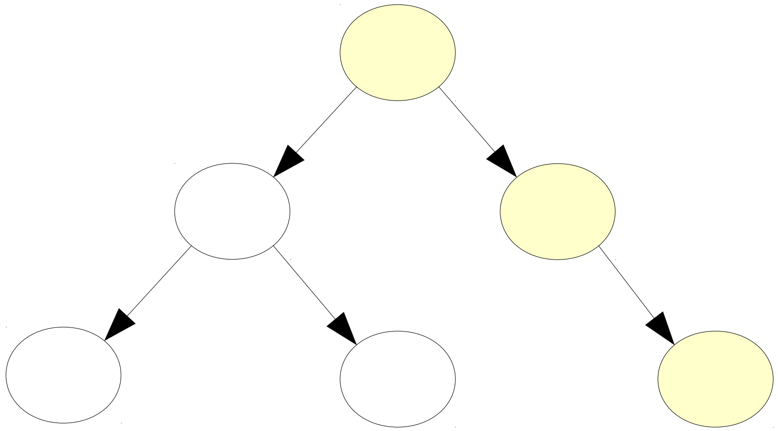
- With the DSE approach, we can only go through one single path at a time.



Paths discovered at the 2^{nd} iteration

Symbolic Execution

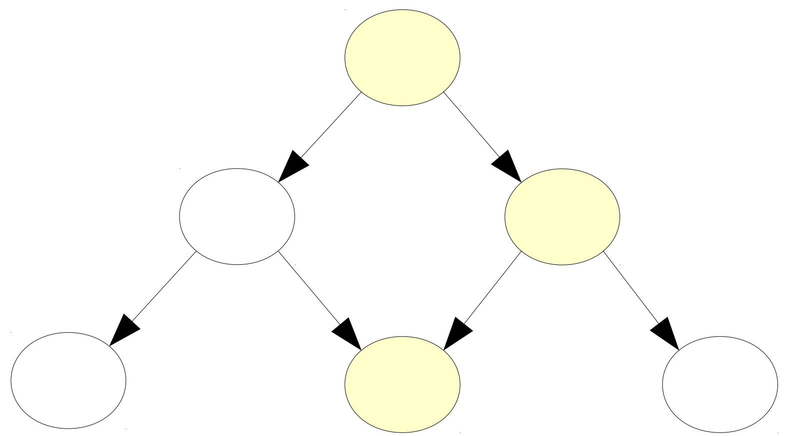
- With the DSE approach, we can only go through one single path at a time.



Paths discovered at the 3rd iteration

Symbolic Execution

- With the DSE approach, we can only go through one single path at a time.

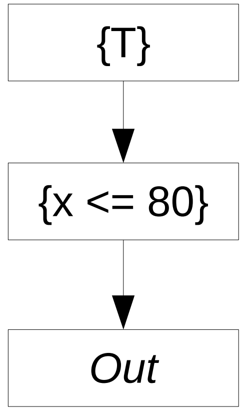


Paths discovered at the 4^{th} iteration

Symbolic Execution

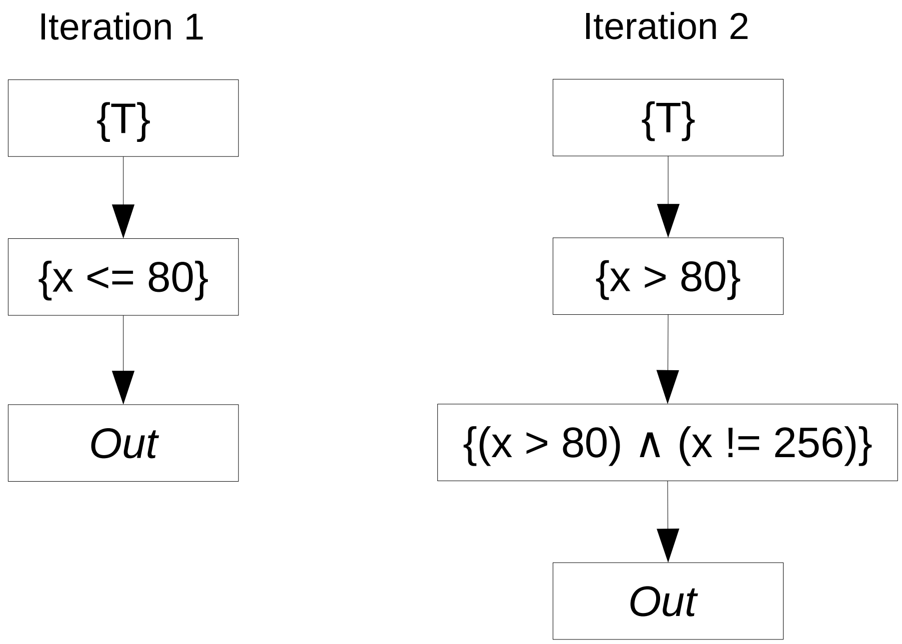
- In this example, the DSE approach will iterate 3 times and keep the formula for all paths

Iteration 1



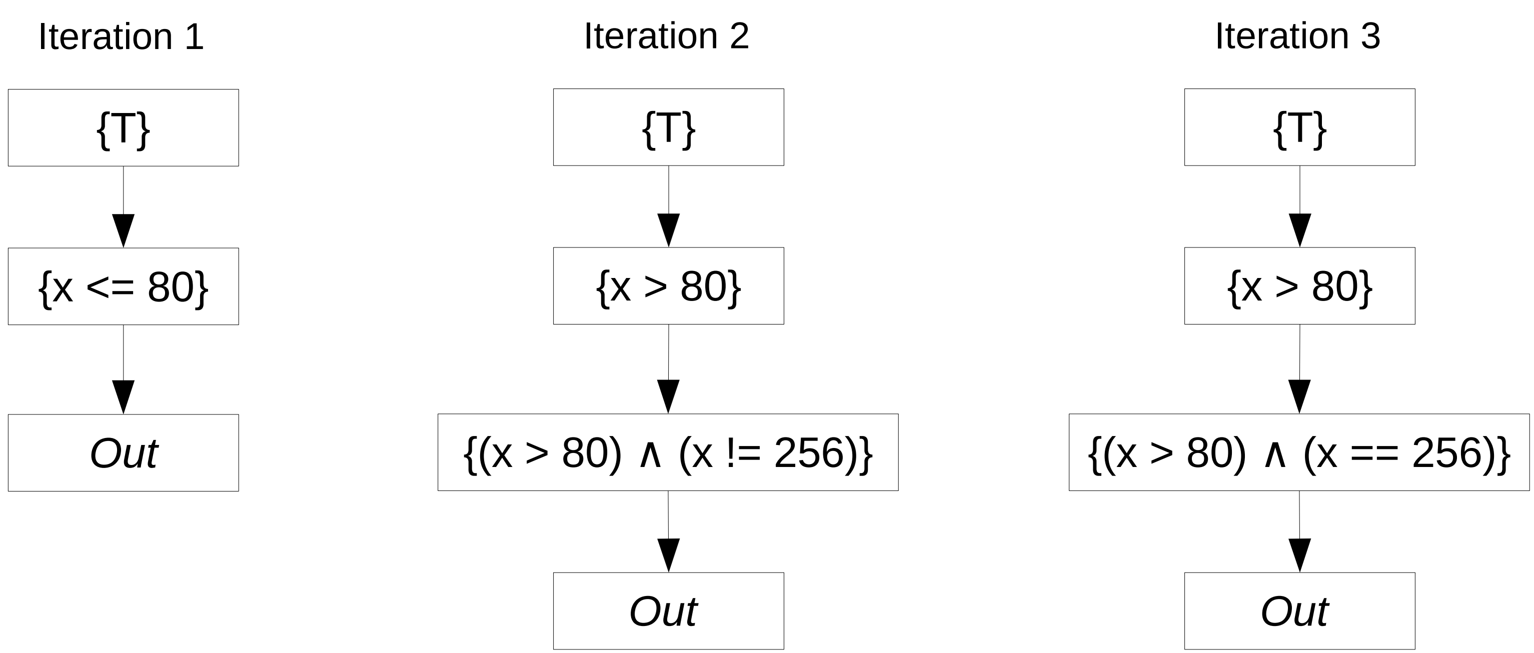
Symbolic Execution

- In this example, the DSE approach will iterate 3 times and keep the formula for all paths



Symbolic Execution

- In this example, the DSE approach will iterate 3 times and keep the formula for all paths



Introduction
Theorem Prover

Theorem Prover

- Used to prove if an equation is satisfiable or not
  - Example with a simple equation with two unknown values

```
$ cat ./ex.py
from z3 import *

x = BitVec('x', 32)
y = BitVec('y', 32)

s = Solver()
s.add((x ^ 0x55) + (3 - (y * 12)) == 0x30)
s.check()
print s.model()

$ ./ex.py
[x = 184, y = 16]
```

- Check Axel's previous talk for more information about z3 and theorem prover

Theorem Prover

- Why in our case do we use a theorem prover?
  - To check if a path constraint (PC) can be solved and with which model
  - Example with the previous code (slide 22)
    - What value can hold the variable *'x'* to take the *“return false”* path?

```
>>> from z3 import *
>>> x = BitVec('x', 32)
>>> s = Solver()
>>> s.add(Or(x <= 80, And(x > 80, x != 256)))
>>> s.check()
sat
>>> s.model()
[x = 0]
```

OK, now that the introduction is over, let's start the talk !

Objective?

- Objective: Cover a function using a DSE approach

- To do that, we will:
  1. Target a function in memory
  2. Setup the context snapshot on this function
  3. Execute this function symbolically
  4. Restore the context and take another path
  5. Repeat this operation until the function is covered

Objective?

- The objective is to cover the *check_password* function
  - Does covering the function mean finding the good password?
    - Yes, we can reach the *return 0* only if we go through all loop iterations

```
char *serial = "\x31\x3e\x3d\x26\x31";

int check_password(char *ptr)
{
  int i = 0;

  while (i < 5){
    if (((ptr[i] - 1) ^ 0x55) != serial[i])
      return 1; /* bad password */
    i++;
  }
  return 0; /* good password */
}
```

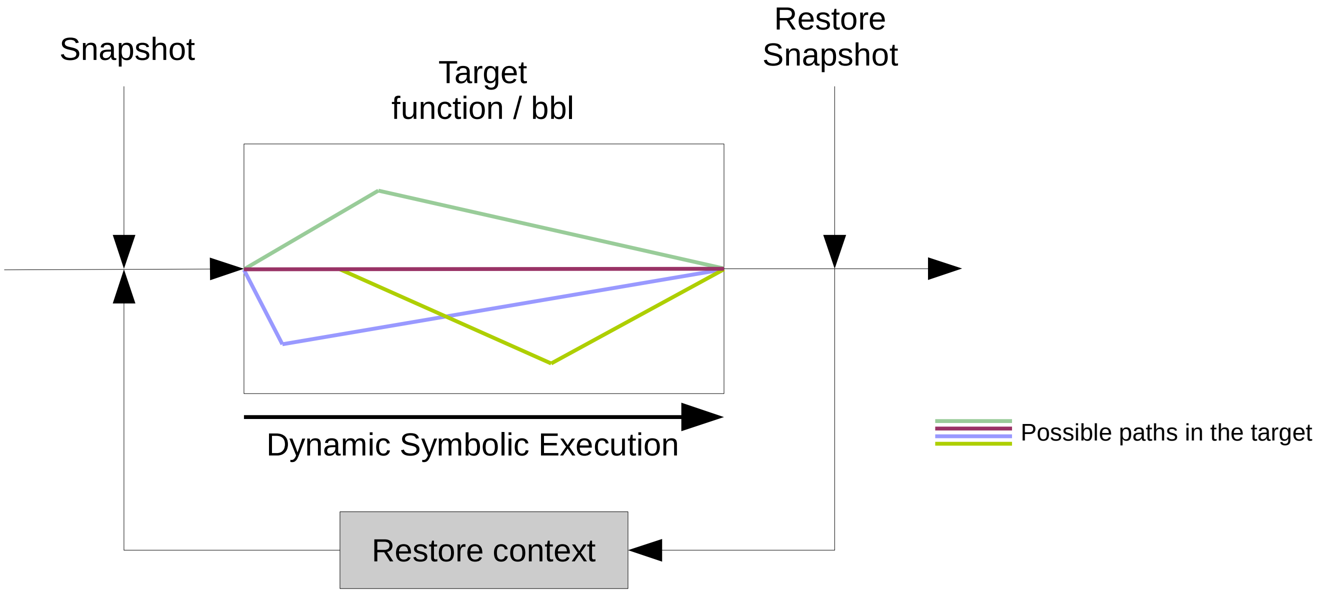
Roadmap

- Save the memory context and the register states (snapshot)
- Taint the *ptr* argument (It is basically our *'x'* of the formula)
- Spread the taint and build the path constraints
  - An operation/statement is an instruction (noted φ_i)
- At the *branch* instruction, use a theorem prover to take the true or the false branch
  - In our case, the goal is to take the false branch (not the *return 1*)
- Restore the memory context and the register states to take another path

Snapshot

Snapshot

- Take a context snapshot at the beginning of the function
- When the function returns, restore the initial context snapshot and go through another path
- Repeat this operation until all the paths are taken



Snapshot

- Use ***PIN_SaveContext()*** to deal with the register states
  - **Save_Context()** only saves register states, not memory
    - We must monitor I/O memory
  - Save context

```
std::cout << "[snapshot]" << std::endl;
PIN_SaveContext(ctx, &snapshot);
```

  - Restore context

```
std::cout << "[restore snapshot]" << std::endl;
PIN_SaveContext(&snapshot, ctx);
restoreMemory();
PIN_ExecuteAt(ctx);
```

Snapshot

- The “restore memory” function looks like this:

```
VOID restoreMemory(void)
{
  list<struct memoryInput>::iterator i;
  for(i = memInput.begin(); i != memInput.end(); ++i){
    *(reinterpret_cast<ADDRINT*>(i->address)) = i->value;
  }
  memInput.clear();
}
```

- The ***memoryInput*** list is filled by monitoring all the ***STORE*** instructions

```
if (INS_OperandCount(ins) > 1 && INS_MemoryOperandIsWritten(ins, 0)){
  INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)WriteMem,
      IARG_ADDRINT, INS_Address(ins),
      IARG_PTR, new string(INS_Disassemble(ins)),
      IARG_UINT32, INS_OperandCount(ins),
      IARG_UINT32, INS_OperandReg(ins, 1),
      IARG_MEMORYOP_EA, 0,
      IARG_END);
}
```

Registers and memory symbolic references

Register references

- A symbolic trace is a sequence of semantic expressions

  $T = (⟦E_1⟧ \wedge ⟦E_2⟧ \wedge ⟦E_3⟧ \wedge ⟦E_4⟧ \wedge \ldots \wedge ⟦E_i⟧)$

- Each expression $⟦E_i⟧ \rightarrow SE_i$ (Symbolic Expression)
- Each SE is translated like this:

  REF_{out} = semantic

  - Where :
    - REF_{out} := unique ID
    - Semantic := $\mathbb{Z}$ | REF_{in} | <<op>>
- A register points on its last reference. Basically, it is close to SSA (Single Static Assignment) but with semantics

Register references

<u>Example:</u>

```
mov eax, 1
add eax, 2
mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
        EAX : -1,
        EBX : -1,
        ECX : -1,
        ...
}
```

```
// Empty set
Symbolic Expression Set {
}
```

Register references

Example:

```
► mov eax, 1            φ0 = 1
  add eax, 2
  mov ebx, eax
```

// All refs initialized to -1
Register Reference Table {
    EAX : φ0,
    EBX : -1,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    <φ0, 1>
}

Register references

Example:

```
mov eax, 1           φ0 = 1
add eax, 2           φ1 = add(φ0, 2)
mov ebx, eax
```

```
// All refs initialized to -1
Register Reference Table {
    EAX : φ1,
    EBX : -1,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <φ1, add(φ0, 2)>,
    <φ0, 1>
}
```

Register references

Example:

```
mov eax, 1            φ0 = 1
add eax, 2            φ1 = add(φ0, 2)
mov ebx, eax          φ2 = φ1
```

```
// All refs initialized to -1
Register Reference Table {
    EAX : φ1,
    EBX : φ2,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <φ2, φ1>,
    <φ1, add(φ0, 2)>,
    <φ0, 1>
}
```

Rebuild the trace with backward analysis

Example:

```
mov eax, 1
add eax, 2
mov ebx, eax
```

→ What is the semantic trace of EBX ?

```
// All refs initialized to -1
Register Reference Table {
      EAX : φ1,
      EBX : φ2,
      ECX : -1,
      ...
}
```

```
// Empty set
Symbolic Expression Set {
      <φ2, φ1>,
      <φ1, add(φ0, 2)>,
      <φ0, 1>
}
```

Rebuild the trace with backward analysis

Example:

```
mov eax, 1
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

```
// All refs initialized to -1
Register Reference Table {
    EAX : φ1,
    EBX : φ2,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <φ2, φ1>,
    <φ1, add(φ0, 2)>,
    <φ0, 1>
}
```

EBX holds the reference **φ2**

Rebuild the trace with backward analysis

Example:

```
mov eax, 1
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

```
// All refs initialized to -1
Register Reference Table {
      EAX : φ1,
      EBX : φ2,
      ECX : -1,
      ...
}
```

```
// Empty set
Symbolic Expression Set {
      <φ2, φ1>,
      <φ1, add(φ0, 2)>,
      <φ0, 1>
}
```

EBX holds the reference **φ2**

What is **φ2** ?

Rebuild the trace with backward analysis

Example:

```
mov eax, 1
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
EAX : φ1,
EBX : **φ2**,
ECX : -1,
...
}

// Empty set
Symbolic Expression Set {
<**φ2**, **φ1**>,
<φ1, add(φ0, 2)>,
<φ0, 1>
}

EBX holds the reference **φ2**

What is **φ2** ?

Reconstruction: EBX = **φ2**

Rebuild the trace with backward analysis

Example:

```
mov eax, 1
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

```
// All refs initialized to -1
Register Reference Table {
    EAX : φ1,
    EBX : φ2,
    ECX : -1,
    ...
}
```

```
// Empty set
Symbolic Expression Set {
    <φ2, φ1>,
    <φ1, add(φ0, 2)>,
    <φ0, 1>
}
```

EBX holds the reference **φ2**

What is **φ2** ?

Reconstruction: EBX = **φ1**

Rebuild the trace with backward analysis

Example:

```
mov eax, 1
add eax, 2
mov ebx, eax
```

→ What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
    EAX : φ1,
    EBX : **φ2**,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    <φ2, φ1>,
    <**φ1, add(φ0, 2)**>, ←
    <φ0, 1>
}

EBX holds the reference **φ2**

What is **φ2** ?

Reconstruction: EBX = **add(φ0, 2)**

Rebuild the trace with backward analysis

Example:

```
mov eax, 1
add eax, 2
mov ebx, eax
```

What is the semantic trace of EBX ?

// All refs initialized to -1
Register Reference Table {
    EAX : φ1,
    EBX : **φ2**,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    <φ2, φ1>,
    <φ1, add(φ0, 2)>,
    <**φ0**, **1**>
}

EBX holds the reference **φ2**

What is **φ2** ?

Reconstruction: EBX = **add(1, 2)**

Follow references over memory

- Assigning a reference for each register is not enough, we must also add references on memory

```
mov dword ptr [rbp-0x4], 0x0
...
mov eax, dword ptr [rbp-0x4]
```

```
push eax
...
pop ebx
```

<u>What do we want to know?</u>

```
eax = 0
```

```
ebx = eax
```

<u>References</u>

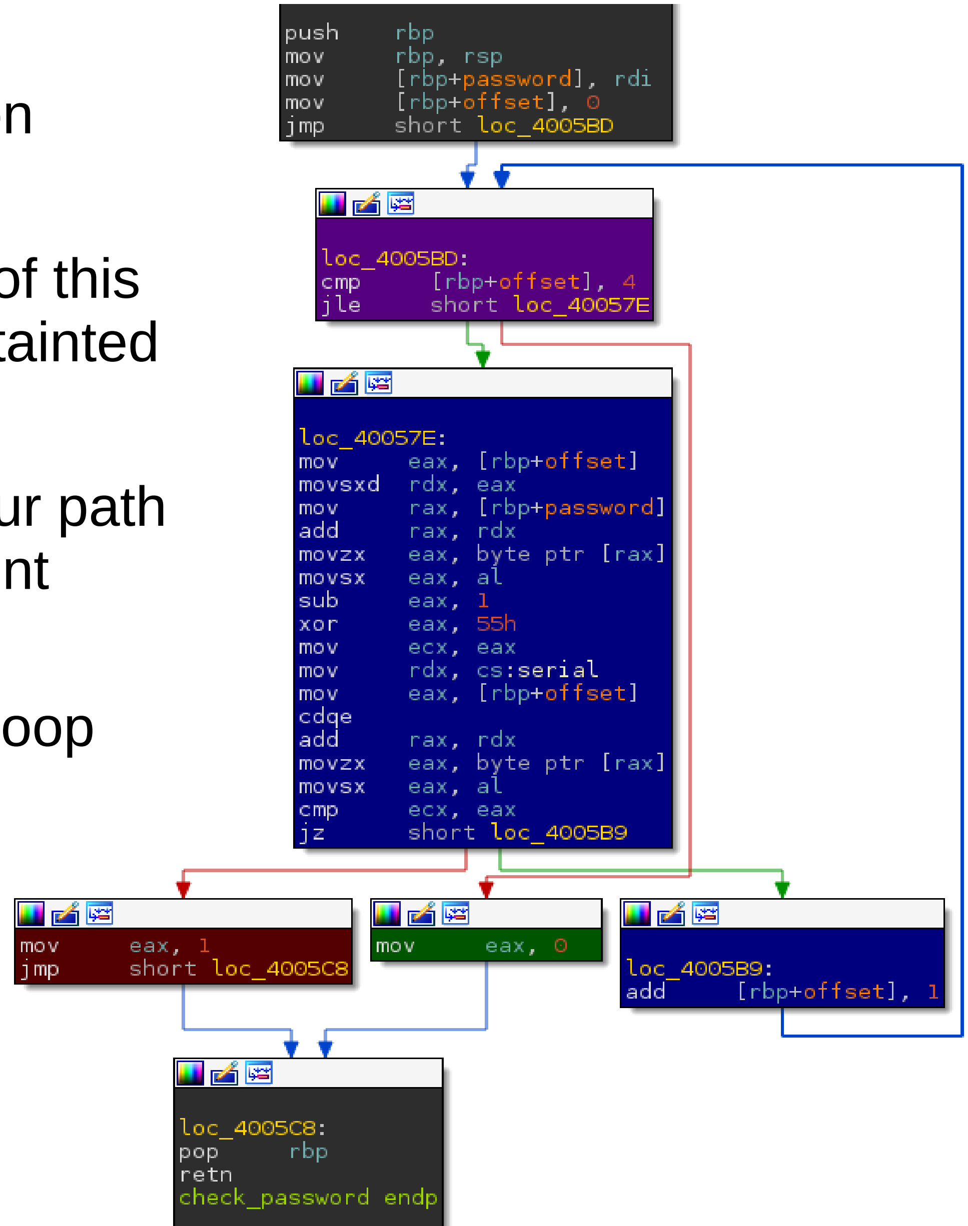
```
φ1 = 0x0
...
φx = φ1
```

```
φ2 = φlast_eax_ref
...
φx = φ2
```

Let's do a DSE on our example

Let's do a DSE on our example

- This is the CFG of the function *check_password*
- RDI holds the first argument of this function. So, RDI points to a tainted area
  - We will follow and build our path constraints only on the taint propagation
- Let's zoom only on the body loop



Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

Empty set

φ_1 = offset

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)

φ2 = SignExt(φ1)

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

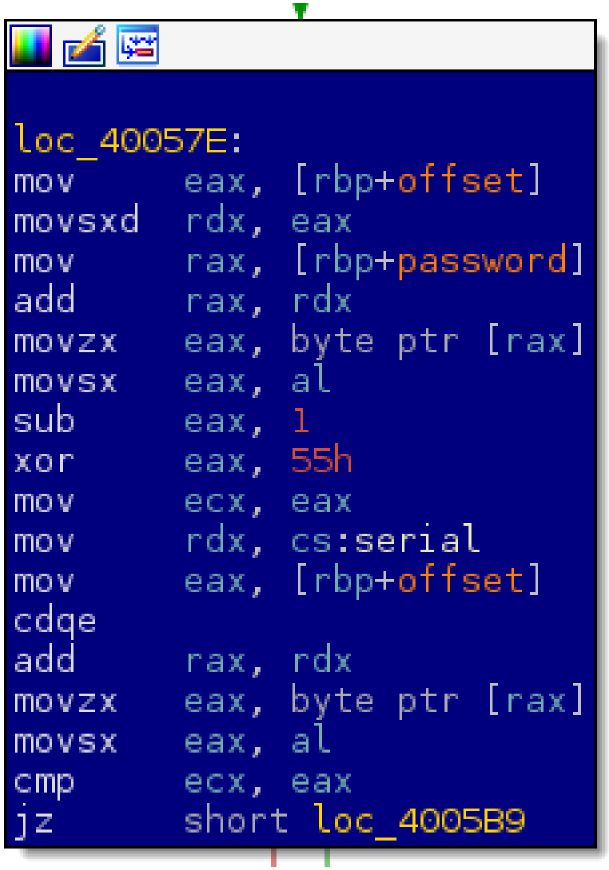
Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ_1 = offset (constant)
φ_2 = SignExt(φ_1)

φ_3 = ptr



Let's do a DSE on our example

- DSE path formula construction

<u>Symbolic Expression Set</u>

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)

φ4 = add(φ3, φ2) →

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)

φ5 = ZeroExt(X)

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)

φ6 = sub(φ5, 1) →

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

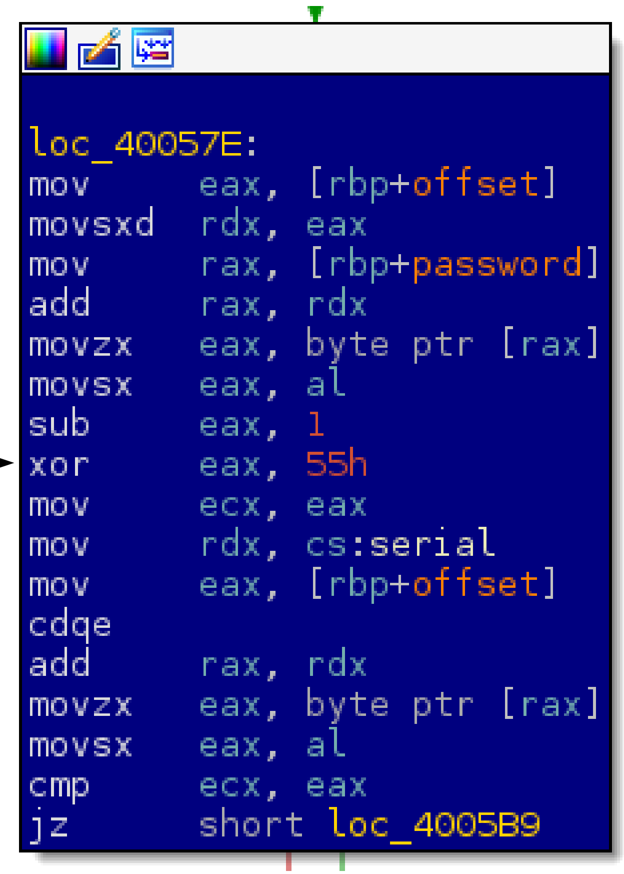
Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(φ5, 1)

φ7 = xor(φ6, 0x55)



Let's do a DSE on our example

- DSE path formula construction

<u>Symbolic Expression Set</u>

$\varphi 1$ = offset (constant)
$\varphi 2$ = SignExt($\varphi 1$)
$\varphi 3$ = ptr (constant)
$\varphi 4$ = add($\varphi 3$, $\varphi 2$)
$\varphi 5$ = ZeroExt(X) (controlled)
$\varphi 6$ = sub($\varphi 5$, 1)
$\varphi 7$ = xor($\varphi 6$, 0x55)

$\varphi 8 = \varphi 7 \longrightarrow$

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

$\varphi1$ = offset (constant)
$\varphi2$ = SignExt($\varphi1$)
$\varphi3$ = ptr (constant)
$\varphi4$ = add($\varphi3$, $\varphi2$)
$\varphi5$ = ZeroExt(X) (controlled)
$\varphi6$ = sub($\varphi5$, 1)
$\varphi7$ = xor($\varphi6$, 0x55)
$\varphi8$ = $\varphi7$

$\varphi9$ = ptr

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

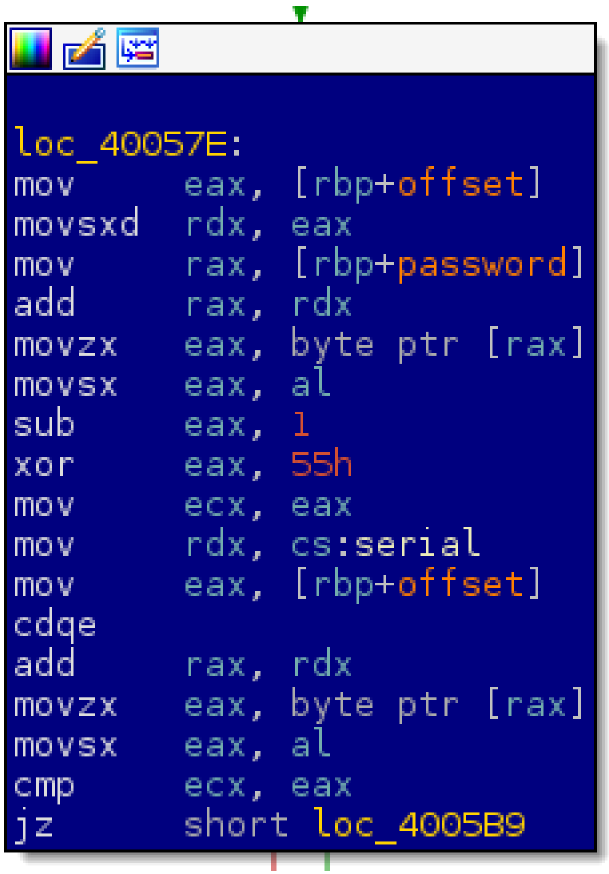
Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(φ5, 1)
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)

φ10 = offset



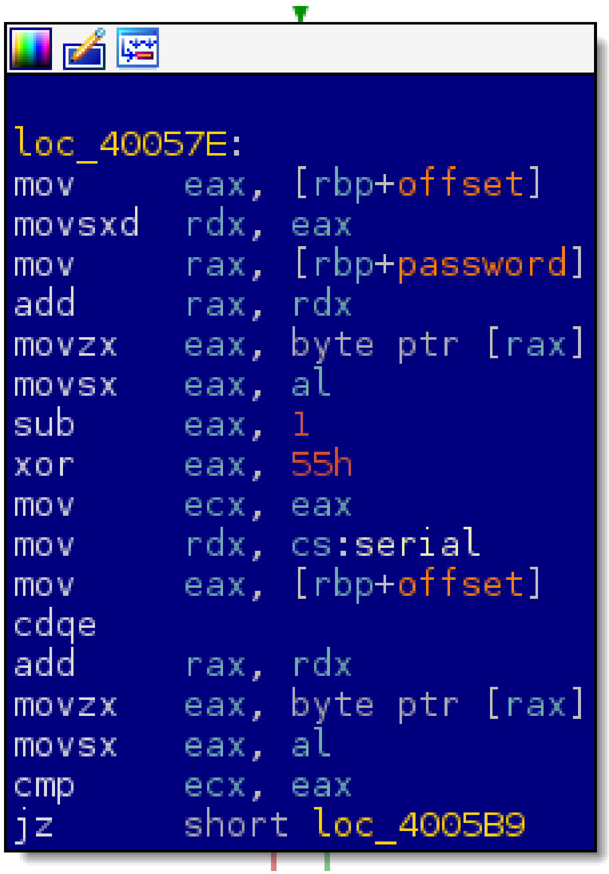
Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(φ5, 1)
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset

φ11 = add(φ10, φ9)



Let's do a DSE on our example

- DSE path formula construction

<u>Symbolic Expression Set</u>

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(φ5, 1)
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)

φ12 = constant

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(φ5, 1)
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant

φ13 = φ12

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

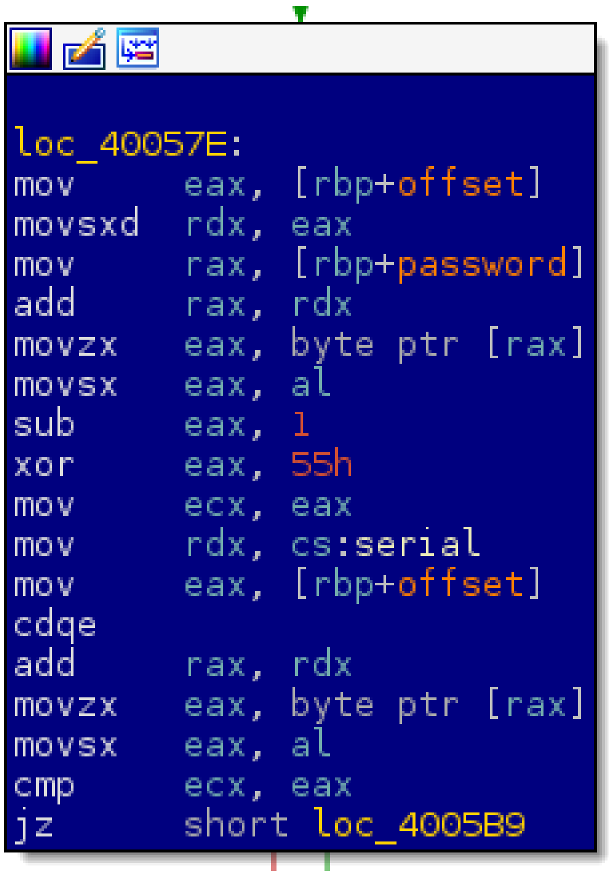
Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(φ5, 1)
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

φ14 = cmp(φ8, φ13)



Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(φ5, 1)
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

φ14 = cmp(φ8, φ13)

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) ← X is controllable
φ6 = sub(φ5, 1)
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

φ14 = cmp(φ8, φ13) →

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1) ← Spread
φ7 = xor(φ6, 0x55)
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

φ14 = cmp(φ8, φ13) →

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

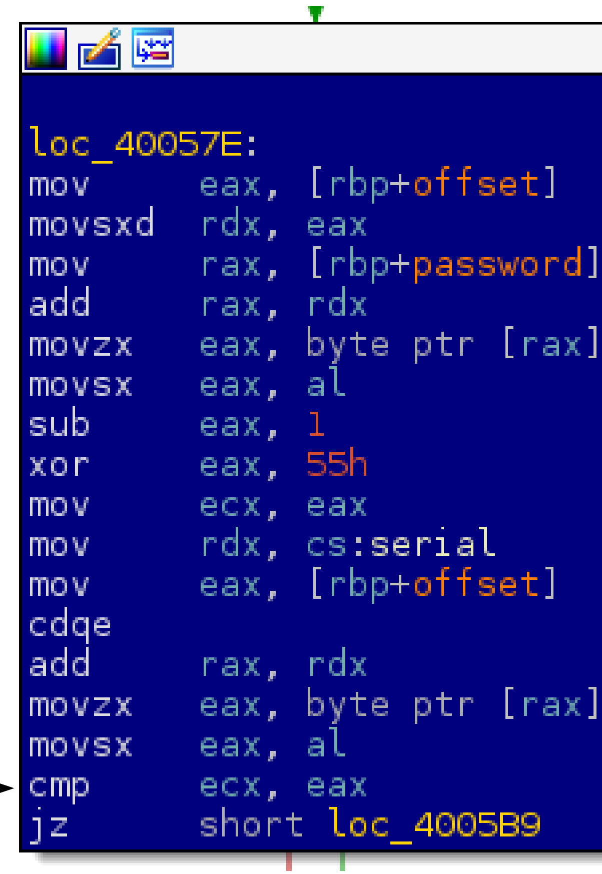
Let's do a DSE on our example

- OK. Now, what the user can control?

<u>Symbolic Expression Set</u>

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55) ← Spread
φ8 = φ7
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

<u>φ14 = cmp(φ8, φ13)</u> →



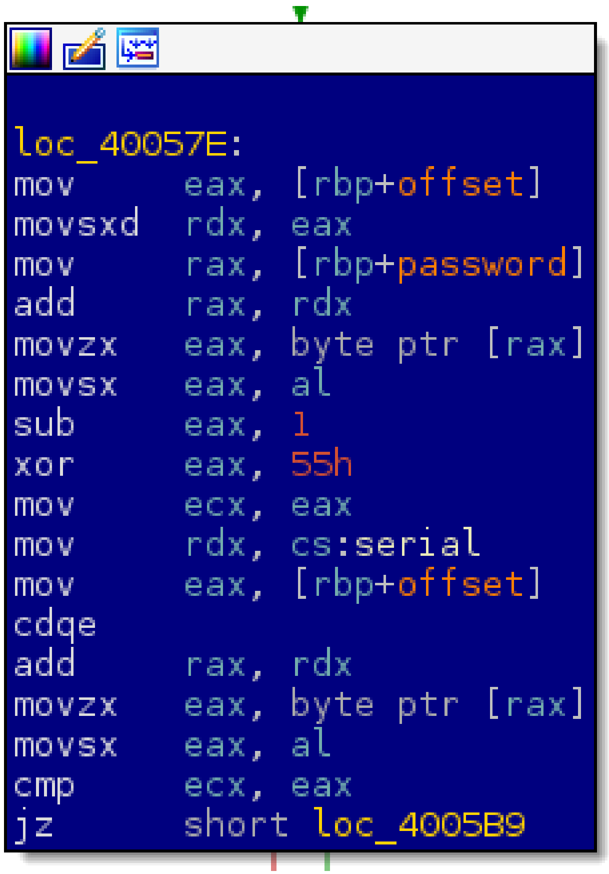
Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7** ← Spread
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

φ14 = cmp(φ8, φ13) →



Let's do a DSE on our example

- OK. Now, what the user can control?

<u>Symbolic Expression Set</u>

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

Controllable

<u>φ14 = cmp(**φ8**, φ13)</u>

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

Controllable

φ14 = cmp(**φ8**, φ13)

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Formula reconstruction: cmp(**φ8**, φ13)

Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

Controllable

φ14 = cmp(**φ8**, φ13)

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Formula reconstruction: cmp(**φ7**, φ13)

Let's do a DSE on our example

- OK. Now, what the user can control?

<u>Symbolic Expression Set</u>

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

Controllable

<u>φ14 = cmp(**φ8**, φ13)</u>

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

<u>Formula reconstruction:</u> cmp(xor(**φ6**, 0x55), φ13)

Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

Controllable

φ14 = cmp(**φ8**, φ13)

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Formula reconstruction: cmp(xor(sub(**φ5**, 1), 0x55), φ13)

Let's do a DSE on our example

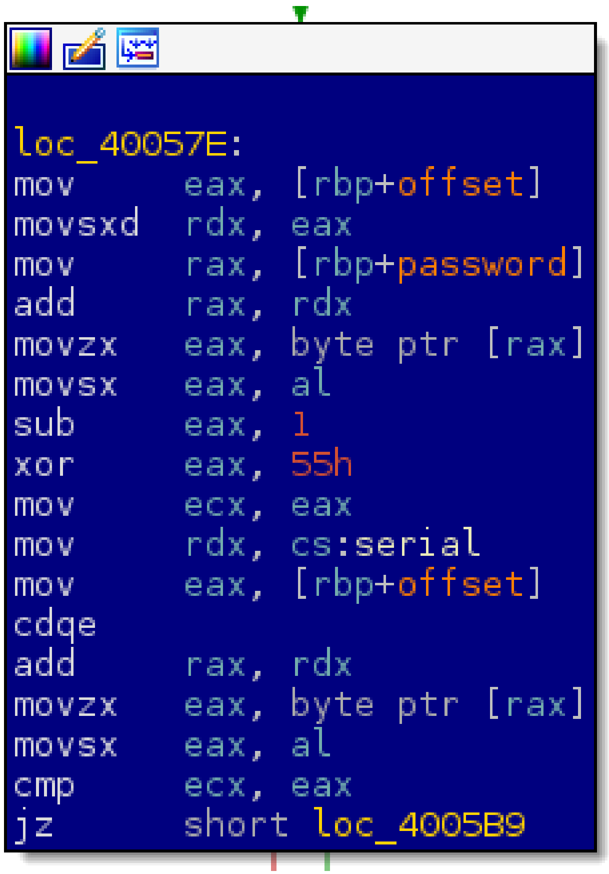
- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = φ12

Controllable

φ14 = cmp(**φ8**, φ13)



Formula reconstruction: cmp(xor(sub(ZeroExt(**X**), 1), 0x55), φ13)

Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = ZeroExt(X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = constant
φ13 = **φ12**

Reconstruction

φ14 = cmp(**φ8**, **φ13**)

```
loc_40057E:
mov     eax, [rbp+offset]
movsxd  rdx, eax
mov     rax, [rbp+password]
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
sub     eax, 1
xor     eax, 55h
mov     ecx, eax
mov     rdx, cs:serial
mov     eax, [rbp+offset]
cdqe
add     rax, rdx
movzx   eax, byte ptr [rax]
movsx   eax, al
cmp     ecx, eax
jz      short loc_4005B9
```

Formula reconstruction: cmp(xor(sub(ZeroExt(**X**), 1), 0x55), **φ12**)

Let's do a DSE on our example

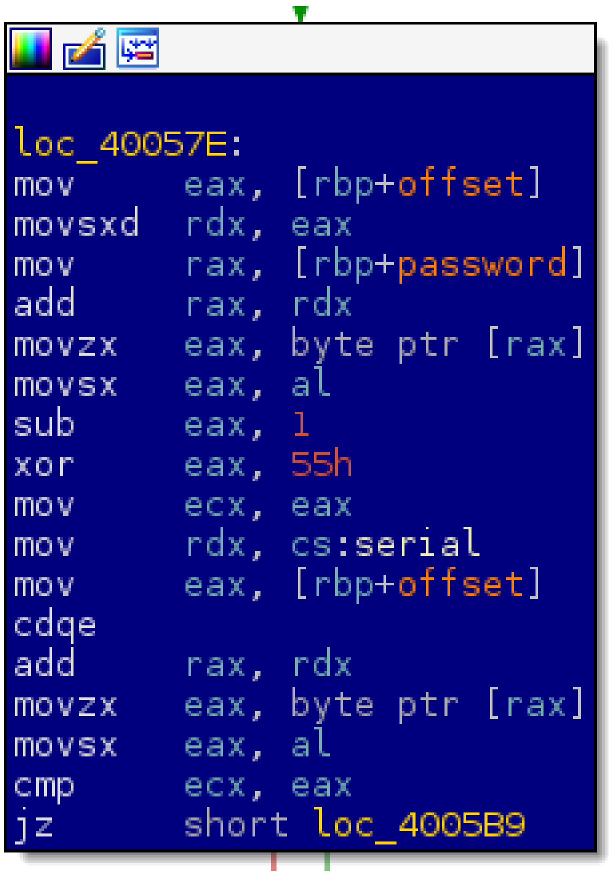
- OK. Now, what the user can control?

Symbolic Expression Set

φ1 = offset (constant)
φ2 = SignExt(φ1)
φ3 = ptr (constant)
φ4 = add(φ3, φ2)
φ5 = (X) (controlled)
φ6 = sub(**φ5**, 1)
φ7 = xor(**φ6**, 0x55)
φ8 = **φ7**
φ9 = ptr (constant)
φ10 = offset
φ11 = add(φ10, φ9)
φ12 = **constant**
φ13 = **φ12**

Reconstruction

φ14 = cmp(**φ8**, **φ13**)



Formula reconstruction: cmp(xor(sub(ZeroExt(**X**), 1), 0x55), **constant**)

Formula reconstruction

- Formula reconstruction: cmp(xor(sub(ZeroExt(**X**) 1), 0x55), **constant**)
  - The **constant** is known at runtime : 0x31 is the constant for the first iteration
- It is time to use Z3

```
>>> from z3 import *
>>> x = BitVec('x', 8)
>>> s = Solver()
>>> s.add(((ZeroExt(32, x) - 1) ^ 0x55) == 0x31)
>>> s.check()
Sat
>>> s.model()
[x = 101]
>>> chr(101)
'e'
```

- To take the true branch the first character of the password must be 'e'.

What path to chose ?

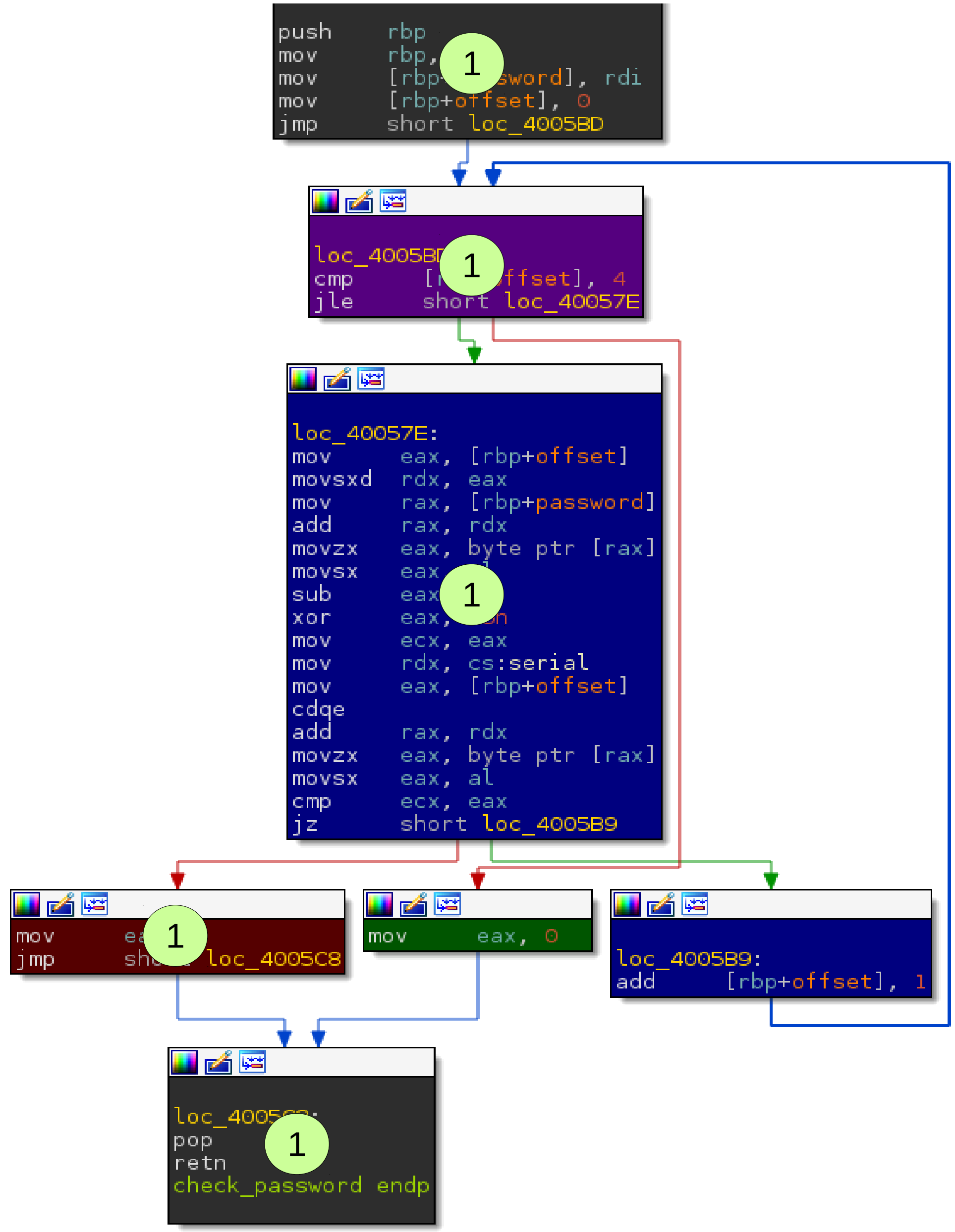
- At this point we got the choice to take the true or the false branch by inverting the formula

```
False = ((x – 1) ⊻ 0x55) != 0x31
True  = ((x – 1) ⊻ 0x55) == 0x31
```

- In our case we must take the true branch to go through the second loop iteration
  - Then, we repeat the same operation until the loop is over

We repeat this operation until all the paths are covered

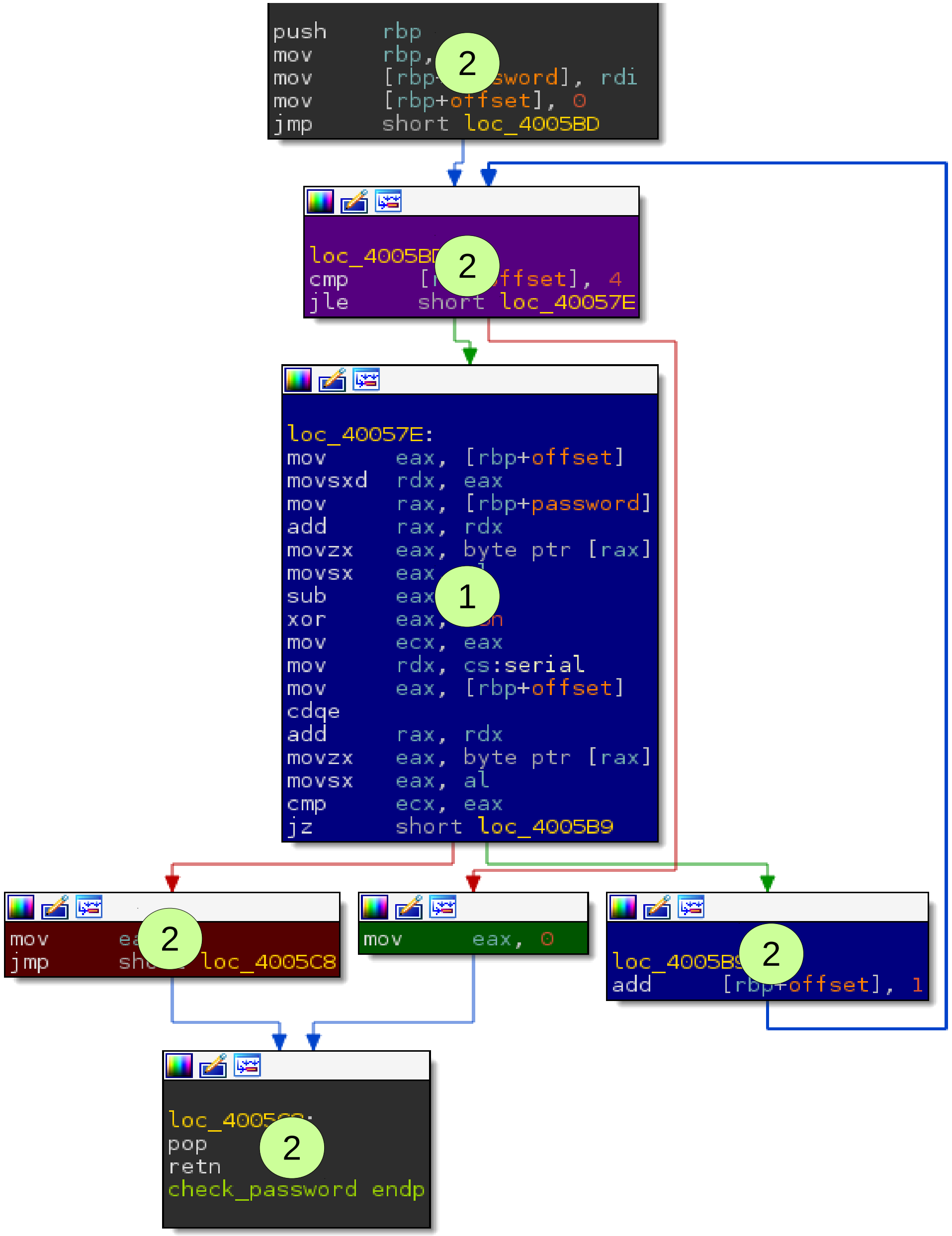
1 $(((\mathbf{x1} - 1) \veebar \text{0x55}) \mathrel{!=} \text{0x31})$



We repeat this operation until all the paths are covered

1 $(((x1 - 1) \veebar 0x55) \mathrel{!=} 0x31)$

2 $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
$(((x2 - 1) \veebar 0x55) \mathrel{!=} 0x3e)$

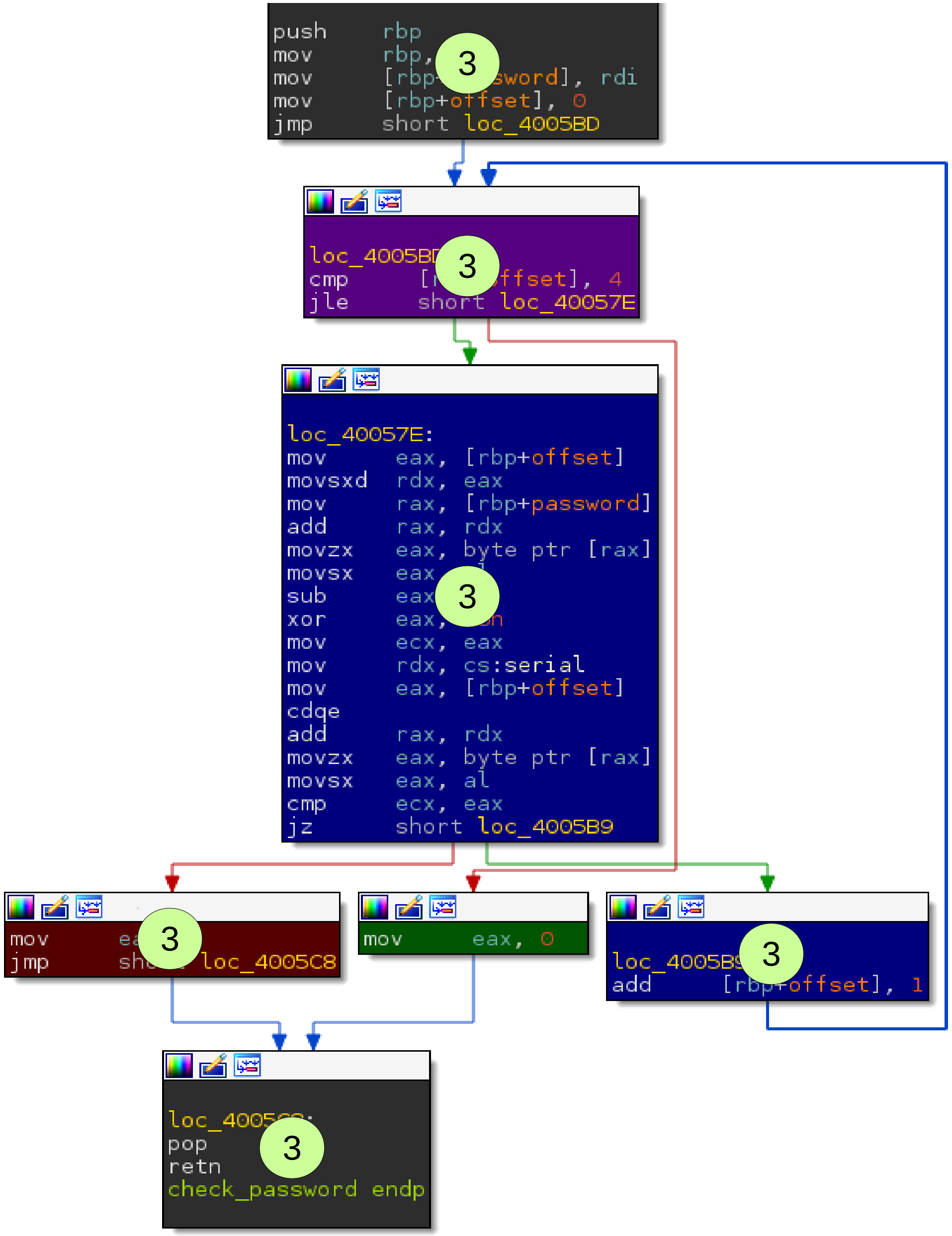


We repeat this operation until all the paths are covered

1 $(((x1 - 1) \veebar 0x55) \mathrel{!=} 0x31)$

2 $(((x1 - 1) \veebar 0x55) == 0x31) \land$
$(((x2 - 1) \veebar 0x55) \mathrel{!=} 0x3e)$

3 $(((x1 - 1) \veebar 0x55) == 0x31) \land$
$(((x2 - 1) \veebar 0x55) == 0x3e) \land$
$(((x3 - 1) \veebar 0x55) \mathrel{!=} 0x3d)$



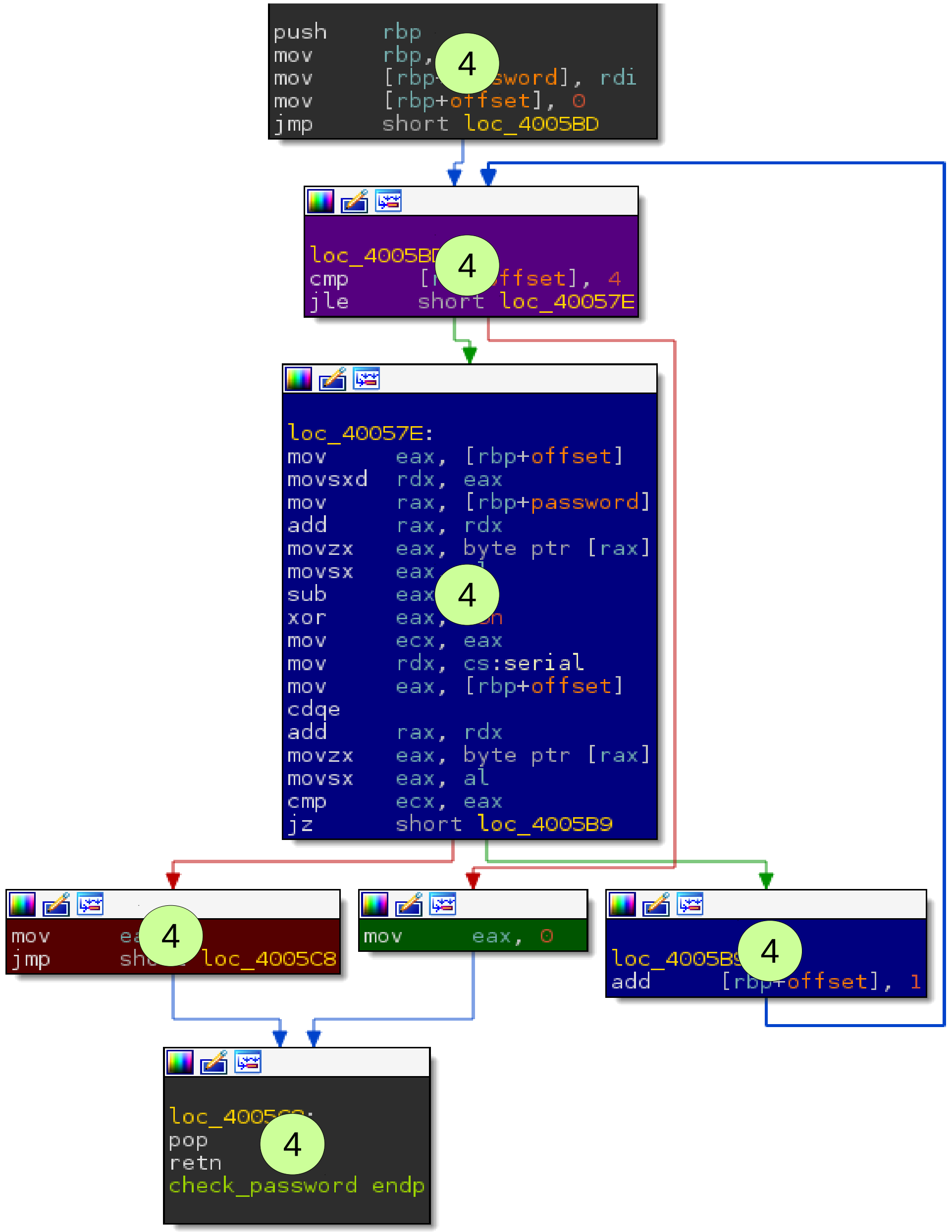
We repeat this operation until all the paths are covered

1. $(((x1 - 1) \veebar 0x55) != 0x31)$

2. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) != 0x3e)$

3. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
   $(((x3 - 1) \veebar 0x55) != 0x3d)$

4. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
   $(((x3 - 1) \veebar 0x55) == 0x3d) \wedge$
   $(((x4 - 1) \veebar 0x55) != 0x26))$



We repeat this operation until all the paths are covered

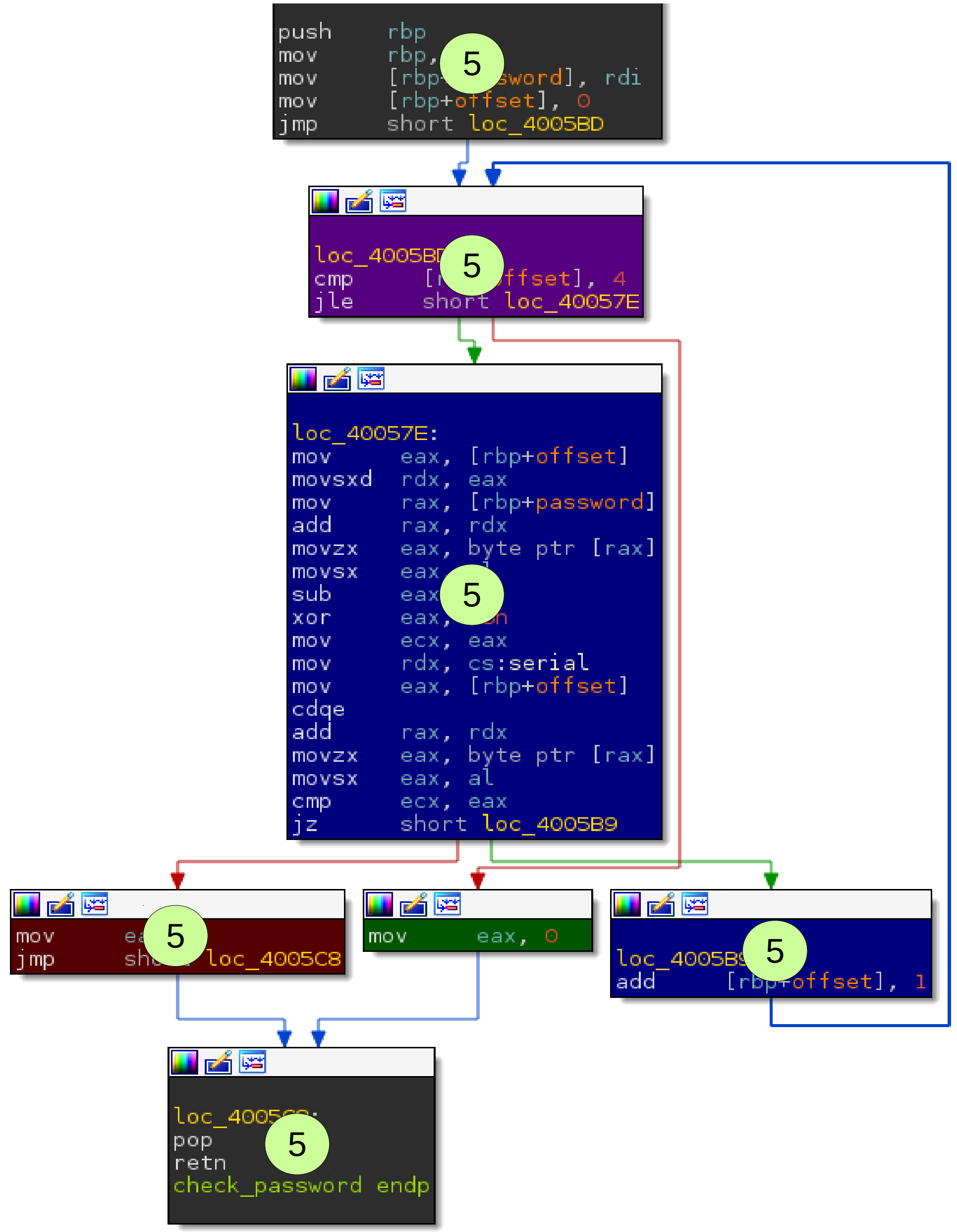
1 $(((x1 - 1) \veebar 0x55) \neq 0x31)$

2 $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
$(((x2 - 1) \veebar 0x55) \neq 0x3e)$

3 $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
$(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
$(((x3 - 1) \veebar 0x55) \neq 0x3d)$

4 $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
$(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
$(((x3 - 1) \veebar 0x55) == 0x3d) \wedge$
$(((x4 - 1) \veebar 0x55) \neq 0x26))$

5 $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
$(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
$(((x3 - 1) \veebar 0x55) == 0x3d) \wedge$
$(((x4 - 1) \veebar 0x55) == 0x26) \wedge$
$(((x5 - 1) \veebar 0x55) \neq 0x31)$



We repeat this operation until all the paths are covered

1. $(((x1 - 1) \veebar 0x55) \neq 0x31)$

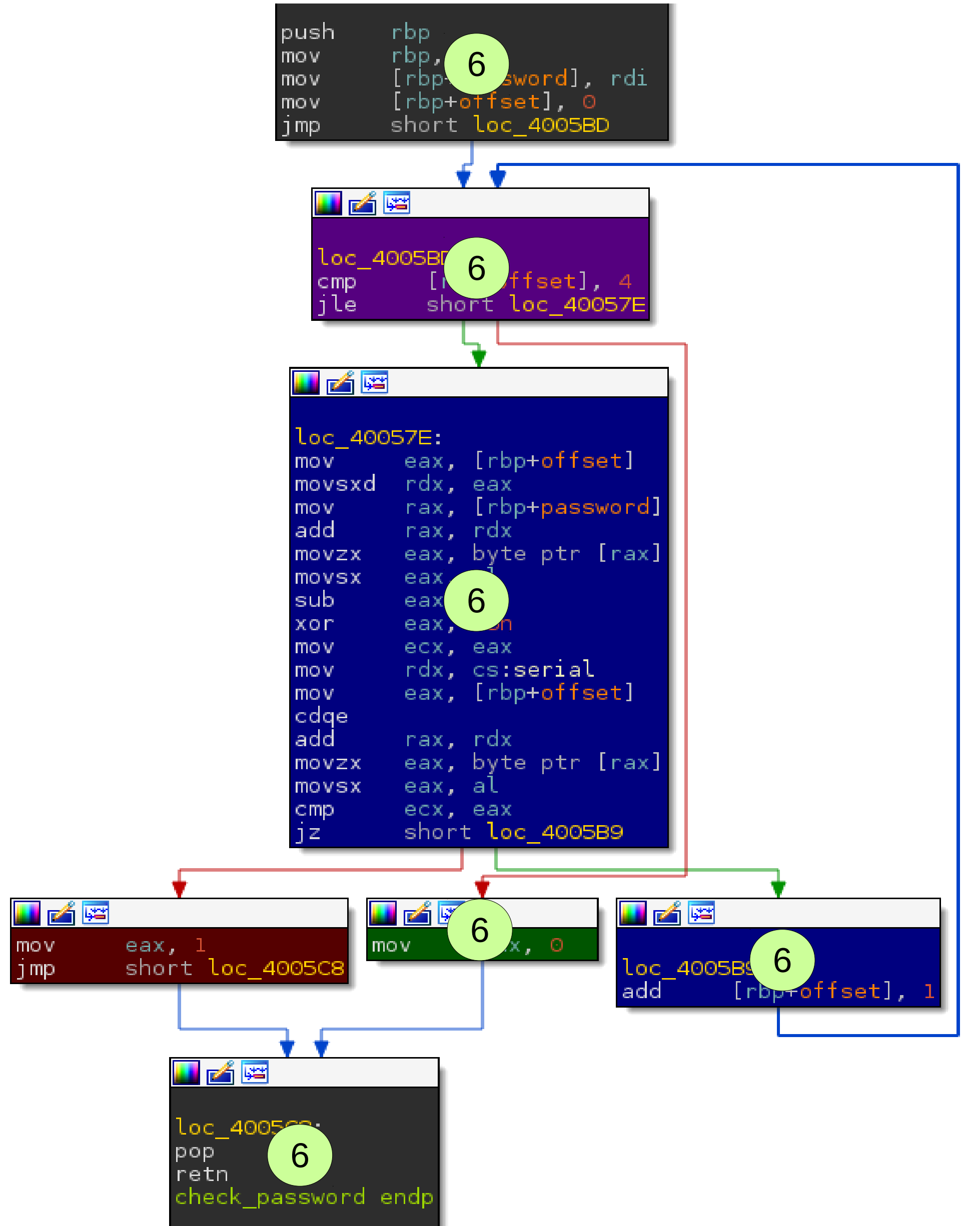
2. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) \neq 0x3e)$

3. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
   $(((x3 - 1) \veebar 0x55) \neq 0x3d)$

4. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
   $(((x3 - 1) \veebar 0x55) == 0x3d) \wedge$
   $(((x4 - 1) \veebar 0x55) \neq 0x26))$

5. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
   $(((x3 - 1) \veebar 0x55) == 0x3d) \wedge$
   $(((x4 - 1) \veebar 0x55) == 0x26) \wedge$
   $(((x5 - 1) \veebar 0x55) \neq 0x31)$

6. $(((x1 - 1) \veebar 0x55) == 0x31) \wedge$
   $(((x2 - 1) \veebar 0x55) == 0x3e) \wedge$
   $(((x3 - 1) \veebar 0x55) == 0x3d) \wedge$
   $(((x4 - 1) \veebar 0x55) == 0x26) \wedge$
   $(((x5 - 1) \veebar 0x55) == 0x31)$



Formula to return 0 or 1


'Dafuq Slide ?!

- The complete formula to *return 0* is:
  - $\beta_i = ((((\mathbf{x1} - 1) \veebar 0x55) == 0x31) \land (((\mathbf{x2} - 1) \veebar 0x55) == 0x3e) \land (((\mathbf{x3} - 1) \veebar 0x55) == 0x3d) \land (((\mathbf{x4} - 1) \veebar 0x55) == 0x26) \land (((\mathbf{x5} - 1) \veebar 0x55) == 0x31))$
    - Where **x1**, **x2**, **x3**, **x4** and **x5** are five variables controlled by the user inputs

- The complete formula to *return 1* is:
  - $\beta_{(i+1)} = ((((\mathbf{x1} - 1) \veebar 0x55) == 0x31) \land (((\mathbf{x2} - 1) \veebar 0x55) == 0x3e) \land (((\mathbf{x3} - 1) \veebar 0x55) == 0x3d) \land (((\mathbf{x4} - 1) \veebar 0x55) == 0x26) \land (((\mathbf{x5} - 1) \veebar 0x55) != 0x31) \lor (((\mathbf{x1} - 1) \veebar 0x55) == 0x31) \land (((\mathbf{x2} - 1) \veebar 0x55) == 0x3e) \land (((\mathbf{x3} - 1) \veebar 0x55) == 0x3d) \land (((\mathbf{x4} - 1) \veebar 0x55) != 0x26)) \lor (((\mathbf{x1} - 1) \veebar 0x55) == 0x31) \land (((\mathbf{x2} - 1) \veebar 0x55) == 0x3e) \land (((\mathbf{x3} - 1) \veebar 0x55) != 0x3d) \lor (((\mathbf{x1} - 1) \veebar 0x55) == 0x31) \land (((\mathbf{x2} - 1) \veebar 0x55) != 0x3e) \lor (((\mathbf{x1} - 1) \veebar 0x55) != 0x31))$
    - Where **x1**, **x2**, **x3**, **x4** and **x5** are five variables controlled by the user inputs

Generate a concrete Value to return 0

'Dafuq Slide ?!

- The complete formula to *return 0* is:
  - $\beta_i = ((((\mathbf{x1} - 1) \veebar 0x55) == 0x31) \wedge (((\mathbf{x2} - 1) \veebar 0x55) == 0x3e) \wedge (((\mathbf{x3} - 1) \veebar 0x55) == 0x3d) \wedge (((\mathbf{x4} - 1) \veebar 0x55) == 0x26) \wedge (((\mathbf{x5} - 1) \veebar 0x55) == 0x31))$

- The concrete value generation using z3

```
>>> from z3 import *
>>> x1, x2, x3, x4, x5 = BitVecs('x1 x2 x3 x4 x5', 8)
>>> s = Solver()
>>> s.add(And((((x1 - 1) ^ 0x55) == 0x31), (((x2 - 1) ^ 0x55) == 0x3e), (((x3 -
1) ^ 0x55) == 0x3d), (((x4 - 1) ^ 0x55) == 0x26), (((x5 - 1) ^ 0x55) == 0x31)))
>>> s.check()
sat
>>> s.model()
[x3 = 105, x2 = 108, x1 = 101, x4 = 116, x5 = 101]
>>> print chr(101), chr(108), chr(105), chr(116), chr(101)
e l i t e
>>>
```

Generate a concrete Value to return 1

'Dafuq Slide ?!

- The complete formula to *return 1* is:
  - $\beta_{(i+1)} = ((((x1 - 1) \veebar 0x55) == 0x31) \land (((x2 - 1) \veebar 0x55) == 0x3e) \land (((x3 - 1) \veebar 0x55) == 0x3d) \land (((x4 - 1) \veebar 0x55) == 0x26) \land (((x5 - 1) \veebar 0x55) \mathrel{!=} 0x31) \lor (((x1 - 1) \veebar 0x55) == 0x31) \land (((x2 - 1) \veebar 0x55) == 0x3e) \land (((x3 - 1) \veebar 0x55) == 0x3d) \land (((x4 - 1) \veebar 0x55) \mathrel{!=} 0x26)) \lor (((x1 - 1) \veebar 0x55) == 0x31) \land (((x2 - 1) \veebar 0x55) == 0x3e) \land (((x3 - 1) \veebar 0x55) \mathrel{!=} 0x3d) \lor (((x1 - 1) \veebar 0x55) == 0x31) \land (((x2 - 1) \veebar 0x55) \mathrel{!=} 0x3e) \lor (((x1 - 1) \veebar 0x55) \mathrel{!=} 0x31))$

- The concrete value generation using z3

```
>>> s.add(Or(And((((x1 - 1) ^ 0x55) == 0x31), (((x2 - 1) ^ 0x55) == 0x3e), (((x3
- 1) ^ 0x55) == 0x3d), (((x4 - 1) ^ 0x55) == 0x26), (((x5 - 1) ^ 0x55) != 0x31)),
And((((x1 - 1) ^ 0x55) == 0x31),(((x2 - 1) ^ 0x55) == 0x3e),(((x3 - 1) ^ 0x55) ==
0x3d),(((x4 - 1) ^ 0x55) != 0x26)), And((((x1 - 1) ^ 0x55) == 0x31),(((x2 - 1) ^
0x55) == 0x3e),(((x3 - 1) ^ 0x55) != 0x3d)), And((((x1 - 1) ^ 0x55) == 0x31),
(((x2 - 1) ^ 0x55) != 0x3e)),(((x1 - 1) ^ 0x55) != 0x31)))
>>> s.check()
sat
>>> s.model()
[x3 = 128, x2 = 128, x1 = 8, x5 = 128, x4 = 128]
```

Formula to cover the function check_password

- P represents the set of all the possible paths
- β represents a symbolic path expression
- To cover the function *check_password* we must generate a concrete value for each β in the set P.

$$P = \{\beta_i, \beta_{i+1}, \beta_{i+k}\}$$
$$\forall \beta \in P : E(G(\beta))$$

Where *E* is the execution and *G* the generation of a concrete value from the symbolic expression β.

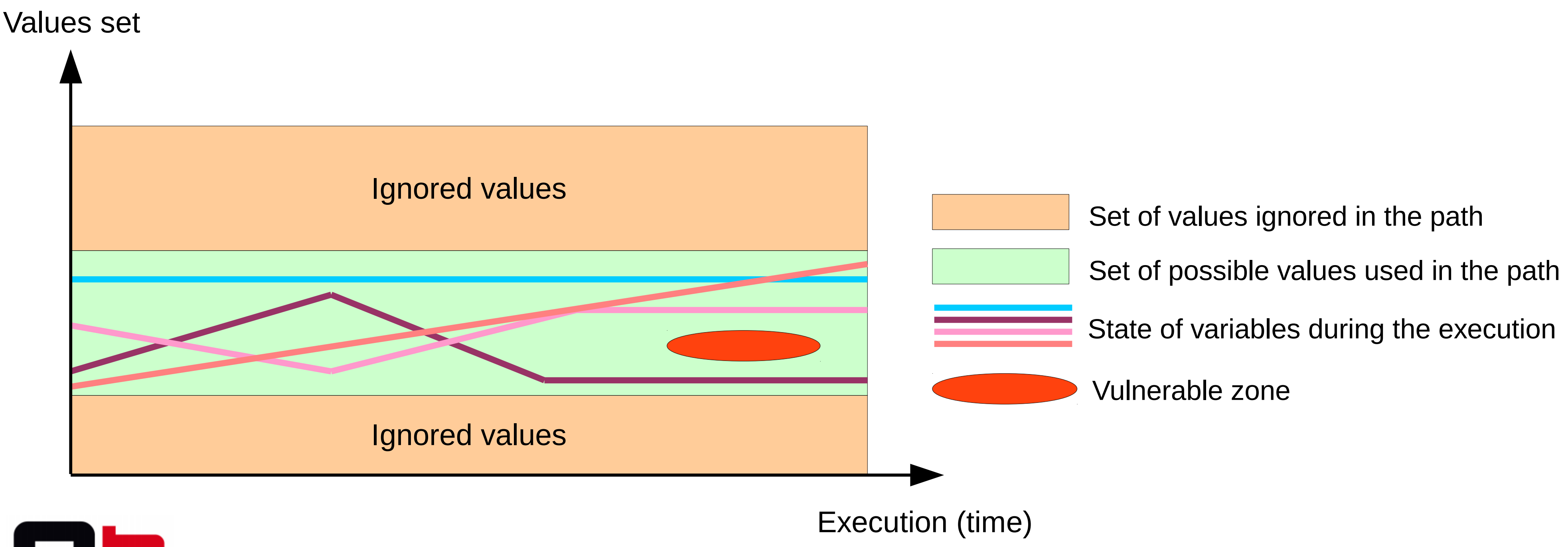
Demo

Video available at https://www.youtube.com/watch?v=1bN-XnpJS2I

Is covering all the paths enough to find vulnerabilities?

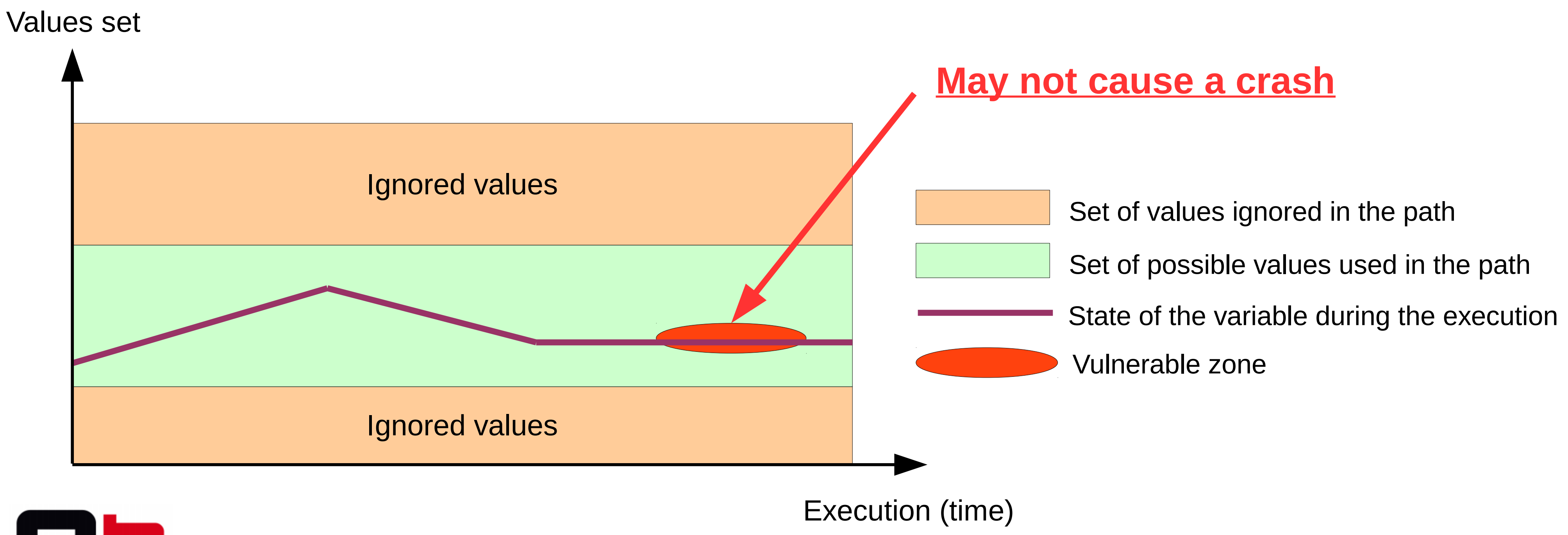
Is covering all the paths enough to find vulnerabilities?

- No! A variable can hold several possible values during the execution and some of these may not trigger any bugs.
- We must generate all concrete values that a path can hold to cover all the possible states.
  - Imply a lot of overload in the worst case
- Below, a Cousot style graph which represents some possible states of a variable during the execution in a path.



A bug may not make the program crash

- Another important point is that a bug may not make the program crash
  - We must implement specific analysis to find specific bugs
    - More detail about these kinds of analysis at my next talk at St'Hack 2015



Conclusion

Conclusion

- Recap:
  - It is possible to cover a targeted function in memory using a DSE approach and memory snapshots.
    - It is also possible to cover all the states of the function but it implies a lot of overload in the worst case

- Future work:
  - Improve the Pin IR
  - Add runtime analysis to find bugs without crashes
    - I will talk about that at the St'Hack 2015 event
  - Simplify an obfuscated trace

Thanks for your attention

- Contact
  - Mail: jsalwan@quarkslab.com
  - Twitter: @JonathanSalwan
- Thanks
  - I would like to thank the security day staff for their invitation and specially Jérémy Fetiveau for the hard work!
  - Then, a big thanks to Ninon Eyrolles, Axel Souchet, Serge Guelton, Jean-Baptiste Bédrune and Aurélien Wailly for their feedbacks.
- Social event
  - Don't forget the doar-e social event after the talks, there are some free beers!