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
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
- Only analyze the executed code
Dynamic Binary Instrumentation

• How does it work in a nutshell?

```plaintext
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

```cpp
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
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
• 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 (□)
Pin and Data Flow Analysis

- Define areas which need to be tagged as controllable
  - For us, this is the environment
    ```c
    int main(int argc, const char *argv[], const char *env[]) {...}
    ```
  - And syscalls
    ```c
    read(fd, buffer, size)
    ```
    Example with sys_read() → For all “byte” in [buffer, buffer+size-1] (Taint(byte))
• Then, spread the taint by monitoring all instructions which read (LOAD) or write (STORE) in the tainted area

```c
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);
}
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]
```

```
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 is the execution of the program using symbolic variables instead of concrete values.

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 pout node?

\[ \text{pout} = \varphi_1 \land ( (\text{pc} \land \varphi_2 \land \varphi_3) \lor (\neg\text{pc} \land \varphi_2' \land \varphi_4) ) \]

\( \varphi = \text{statement / operation} \)
\( \text{pc} = \text{path condition (guard)} \)

Control flow graph
Symbolic Execution

- SSE path formula and statement property

```c
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;
}
```
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

- SSE path formula and statement property

```c
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;
}
```

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

- SSE path formula and statement property

```c
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;
}
```

PC: \{ x1 > 80 \} \{ x2 = y1 \times 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

```c
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: \( \{ x_1 > 80 \land x_2 == 256 \} [x_2 = y_1 \times 2, y_2 = 0] \)
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]
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;
}
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;
}

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

• SSE path formula and statement property
With the DSE approach, we can only go through one single path at a time.

Paths discovered at the 1\textsuperscript{st} iteration
• With the DSE approach, we can only go through one single path at a time.

Paths discovered at the 2\textsuperscript{nd} iteration
Symbolic Execution

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

Paths discovered at the 3\textsuperscript{rd} iteration
With the DSE approach, we can only go through one single path at a time.

Paths discovered at the 4th iteration
In this example, the DSE approach will iterate 3 times and keep the formula for all paths.
In this example, the DSE approach will iterate 3 times and keep the formula for all paths.
In this example, the DSE approach will iterate 3 times and keep the formula for all paths.

**Iteration 1**
- \{T\}
- \{x <= 80\}
- Out

**Iteration 2**
- \{T\}
- \{x > 80\}
- \{(x > 80) \land (x != 256)\}
- Out

**Iteration 3**
- \{T\}
- \{x > 80\}
- \{(x > 80) \land (x == 256)\}
- Out
Introduction
Theorem Prover
Theorem Prover

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

```python
$ 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
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?

```python
>>> 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: 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

```c
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 \textit{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 $\varphi_i$)
- At the \textit{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 \textit{return 1})
- Restore the memory context and the register states to take another path
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
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);
  ```
The “restore memory” function looks like this:

```c
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:

```c
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_MEMORYOP_EA, 0,
        IARG_END);
}
```
Registers and memory symbolic references
A symbolic trace is a sequence of semantic expressions

\[ T = (E_1 \land E_2 \land E_3 \land E_4 \land \ldots \land E_i) \]

Each expression \([E_i] \rightarrow SE_i\) (Symbolic Expression)

Each SE is translated like this:

\[ \text{REF}_{\text{out}} = \text{semantic} \]

- Where:

  - \(\text{REF}_{\text{out}} := \text{unique ID} \)
  - \(\text{Semantic} := \mathbb{Z} \mid \text{REF}_{\text{in}} \mid \text{<<op>>} \)

A register points on its last reference. Basically, it is close to SSA (Single Static Assignment) but with semantics
Example:

```c
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
add eax, 2
mov ebx, eax
```

// All refs initialized to -1
Register Reference Table {
    EAX : $\varphi_0$,
    EBX : -1,
    ECX : -1,
    ...
}

// Empty set
Symbolic Expression Set {
    $<$ $\varphi_0, 1$ $>$
}
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:

```plaintext
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:

```assembly
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(φθ, 2)>,
    <φθ, 1>
}
Rebuild the trace with backward analysis

Example:

```assembly
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:

```assembly
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:

```c
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 : \( \varphi_1 \),
  EBX : \( \varphi_2 \),
  ECX : -1,
  ...
}

// Empty set
Symbolic Expression Set {
  \(<\varphi_2, \varphi_1>\),
  \(<\varphi_1, \text{add}(\varphi_0, 2)>\),
  \(<\varphi_0, 1>\)
}

EBX holds the reference \( \varphi_2 \)

What is \( \varphi_2 \)?

Reconstruction: EBX = \( \varphi_1 \)
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

What is φ₂?

Reconstruction: EBX = add(φ₀, 2)
Rebuild the trace with backward analysis

Example:

```asm
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)
Assigning a reference for each register is not enough, we must also add references on memory.

\[
\begin{align*}
\text{mov dword ptr [rbp-0x4], 0x0} \\
\text{...} \\
\text{mov eax, dword ptr [rbp-0x4]} \\
\text{push eax} \\
\text{...} \\
\text{pop ebx}
\end{align*}
\]

What do we want to know?

\[
\begin{align*}
eax &= 0 \\
\text{ebx} &= \text{eax}
\end{align*}
\]

References

\[
\begin{align*}
\varphi_1 &= 0x0 \\
\text{...} \\
\varphi_x &= \varphi_1 \\
\varphi_2 &= \varphi_{\text{last eax ref}} \\
\text{...} \\
\varphi_x &= \varphi_2
\end{align*}
\]
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

\[ \varphi_1 = \text{offset} \]
Let's do a DSE on our example

- DSE path formula construction

**Symbolic Expression Set**

φ₁ = offset (constant)  \[ φ₂ = \text{SignExt}(φ₁) \]
Let's do a DSE on our example

- **DSE path formula construction**

**Symbolic Expression Set**

\[ \varphi_1 = \text{offset (constant)} \]

\[ \varphi_2 = \text{SignExt}(\varphi_1) \]

\[ \varphi_3 = \text{ptr} \]
Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

\( \varphi_1 = \text{offset (constant)} \)
\( \varphi_2 = \text{SignExt}(\varphi_1) \)
\( \varphi_3 = \text{ptr (constant)} \)
\( \varphi_4 = \text{add}(\varphi_3, \varphi_2) \)

```assembly
loc_40057E:
  mov    eax, [rbp+offset]
  movsx  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

\[ \varphi_1 = \text{offset (constant)} \]
\[ \varphi_2 = \text{SignExt}(\varphi_1) \]
\[ \varphi_3 = \text{ptr (constant)} \]
\[ \varphi_4 = \text{add}(\varphi_3, \varphi_2) \]
\[ \varphi_5 = \text{ZeroExt}(X) \]
Let's do a DSE on our example

- DSE path formula construction

**Symbolic Expression Set**

\[ \varphi_1 = \text{offset (constant)} \]

\[ \varphi_2 = \text{SignExt}(\varphi_1) \]

\[ \varphi_3 = \text{ptr (constant)} \]

\[ \varphi_4 = \text{add}(\varphi_3, \varphi_2) \]

\[ \varphi_5 = \text{ZeroExt}(X) \text{ (controlled)} \]

\[ \varphi_6 = \text{sub}(\varphi_5, 1) \]
Let's do a DSE on our example

- DSE path formula construction

Symbolic Expression Set

\( \varphi_1 = \text{offset (constant)} \)
\( \varphi_2 = \text{SignExt}(\varphi_1) \)
\( \varphi_3 = \text{ptr (constant)} \)
\( \varphi_4 = \text{add}(\varphi_3, \varphi_2) \)
\( \varphi_5 = \text{ZeroExt}(X) \) (controlled)
\( \varphi_6 = \text{sub}(\varphi_5, 1) \)
\( \varphi_7 = \text{xor}(\varphi_6, 0x55) \)
Let's do a DSE on our example

- DSE path formula construction

### Symbolic Expression Set

- \( \phi_1 = \text{offset (constant)} \)
- \( \phi_2 = \text{SignExt}(\phi_1) \)
- \( \phi_3 = \text{ptr (constant)} \)
- \( \phi_4 = \text{add}(\phi_3, \phi_2) \)
- \( \phi_5 = \text{ZeroExt}(X) \) (controlled)
- \( \phi_6 = \text{sub}(\phi_5, 1) \)
- \( \phi_7 = \text{xor}(\phi_6, 0x55) \)

\( \phi_8 = \phi_7 \)
Let's do a DSE on our example

- **DSE path formula construction**

**Symbolic Expression Set**

- $\phi_1 = \text{offset (constant)}$
- $\phi_2 = \text{SignExt}(\phi_1)$
- $\phi_3 = \text{ptr (constant)}$
- $\phi_4 = \text{add}(\phi_3, \phi_2)$
- $\phi_5 = \text{ZeroExt}(X)$ (controlled)
- $\phi_6 = \text{sub}(\phi_5, 1)$
- $\phi_7 = \text{xor}(\phi_6, 0x55)$
- $\phi_8 = \phi_7$
- $\phi_9 = \text{ptr}$

```assembly
loc_40057E:
    mov   eax, [rbp+offset]
    movsx  edx, eax
    mov   rax, [rbp+password]
    add   rax, edx
    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

\[ \varphi_1 = \text{offset (constant)} \]
\[ \varphi_2 = \text{SignExt}(\varphi_1) \]
\[ \varphi_3 = \text{ptr (constant)} \]
\[ \varphi_4 = \text{add}(\varphi_3, \varphi_2) \]
\[ \varphi_5 = \text{ZeroExt}(X) \text{ (controlled)} \]
\[ \varphi_6 = \text{sub}(\varphi_5, 1) \]
\[ \varphi_7 = \text{xor}(\varphi_6, 0x55) \]
\[ \varphi_8 = \varphi_7 \]
\[ \varphi_9 = \text{ptr (constant)} \]
\[ \varphi_{10} = \text{offset} \]
Let's do a DSE on our example

- DSE path formula construction

**Symbolic Expression Set**

\( \varphi_1 = \text{offset (constant)} \)
\( \varphi_2 = \text{SignExt}(\varphi_1) \)
\( \varphi_3 = \text{ptr (constant)} \)
\( \varphi_4 = \text{add}(\varphi_3, \varphi_2) \)
\( \varphi_5 = \text{ZeroExt}(X) \text{ (controlled)} \)
\( \varphi_6 = \text{sub}(\varphi_5, 1) \)
\( \varphi_7 = \text{xor}(\varphi_6, 0x55) \)
\( \varphi_8 = \varphi_7 \)
\( \varphi_9 = \text{ptr (constant)} \)
\( \varphi_{10} = \text{offset} \)

\( \varphi_{11} = \text{add}(\varphi_{10}, \varphi_9) \)
Let's do a DSE on our example

- DSE path formula construction

**Symbolic Expression Set**

\[
\begin{align*}
\phi_1 &= \text{offset (constant)} \\
\phi_2 &= \text{SignExt}(\phi_1) \\
\phi_3 &= \text{ptr (constant)} \\
\phi_4 &= \text{add}(\phi_3, \phi_2) \\
\phi_5 &= \text{ZeroExt}(X) \text{ (controlled)} \\
\phi_6 &= \text{sub}(\phi_5, 1) \\
\phi_7 &= \text{xor}(\phi_6, 0x55) \\
\phi_8 &= \phi_7 \\
\phi_9 &= \text{ptr (constant)} \\
\phi_{10} &= \text{offset} \\
\phi_{11} &= \text{add}(\phi_{10}, \phi_9) \\
\phi_{12} &= \text{constant}
\end{align*}
\]
Let's do a DSE on our example

- DSE path formula construction

**Symbolic Expression Set**

\[
\begin{align*}
\phi_1 &= \text{offset (constant)} \\
\phi_2 &= \text{SignExt}(\phi_1) \\
\phi_3 &= \text{ptr (constant)} \\
\phi_4 &= \text{add}(\phi_3, \phi_2) \\
\phi_5 &= \text{ZeroExt}(X) \text{ (controlled)} \\
\phi_6 &= \text{sub}(\phi_5, 1) \\
\phi_7 &= \text{xor}(\phi_6, 0x55) \\
\phi_8 &= \phi_7 \\
\phi_9 &= \text{ptr (constant)} \\
\phi_{10} &= \text{offset} \\
\phi_{11} &= \text{add}(\phi_{10}, \phi_9) \\
\phi_{12} &= \text{constant}
\end{align*}
\]
Let's do a DSE on our example

- DSE path formula construction

**Symbolic Expression Set**

\[
\begin{align*}
\phi_1 &= \text{offset (constant)} \\
\phi_2 &= \text{SignExt}(\phi_1) \\
\phi_3 &= \text{ptr (constant)} \\
\phi_4 &= \text{add}(\phi_3, \phi_2) \\
\phi_5 &= \text{ZeroExt}(X) \text{ (controlled)} \\
\phi_6 &= \text{sub}(\phi_5, 1) \\
\phi_7 &= \text{xor}(\phi_6, 0x55) \\
\phi_8 &= \phi_7 \\
\phi_9 &= \text{ptr (constant)} \\
\phi_{10} &= \text{offset} \\
\phi_{11} &= \text{add}(\phi_{10}, \phi_9) \\
\phi_{12} &= \text{constant} \\
\phi_{13} &= \phi_{12} \\
\phi_{14} &= \text{cmp}(\phi_8, \phi_{13})
\end{align*}
\]
Let's do a DSE on our example

- OK. Now, what the user can control?

**Symbolic Expression Set**

\[
\begin{align*}
\phi_1 &= \text{offset (constant)} \\
\phi_2 &= \text{SignExt}(\phi_1) \\
\phi_3 &= \text{ptr (constant)} \\
\phi_4 &= \text{add}(\phi_3, \phi_2) \\
\phi_5 &= \text{ZeroExt}(X) \text{ (controlled)} \\
\phi_6 &= \text{sub}(\phi_5, 1) \\
\phi_7 &= \text{xor}(\phi_6, 0x55) \\
\phi_8 &= \phi_7 \\
\phi_9 &= \text{ptr (constant)} \\
\phi_{10} &= \text{offset} \\
\phi_{11} &= \text{add}(\phi_{10}, \phi_9) \\
\phi_{12} &= \text{constant} \\
\phi_{13} &= \phi_{12} \\
\phi_{14} &= \text{cmp}(\phi_8, \phi_{13})
\end{align*}
\]
Let's do a DSE on our example

- OK. Now, what the user can control?

Symbolic Expression Set

\( \phi_1 = \text{offset (constant)} \)
\( \phi_2 = \text{SignExt}(\phi_1) \)
\( \phi_3 = \text{ptr (constant)} \)
\( \phi_4 = \text{add}(\phi_3, \phi_2) \)
\( \phi_5 = \text{ZeroExt}(X) \)
\( \phi_6 = \text{sub}(\phi_5, 1) \)
\( \phi_7 = \text{xor}(\phi_6, 0x55) \)
\( \phi_8 = \phi_7 \)
\( \phi_9 = \text{ptr (constant)} \)
\( \phi_{10} = \text{offset} \)
\( \phi_{11} = \text{add}(\phi_{10}, \phi_9) \)
\( \phi_{12} = \text{constant} \)
\( \phi_{13} = \phi_{12} \)

\( \phi_{14} = \text{cmp}(\phi_8, \phi_{13}) \)

\( X \) is controllable
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)
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)
Let's do a DSE on our example

- OK. Now, what the user can control?

**Symbolic Expression Set**

\[\varphi_1 = \text{offset (constant)}\]
\[\varphi_2 = \text{SignExt}(\varphi_1)\]
\[\varphi_3 = \text{ptr (constant)}\]
\[\varphi_4 = \text{add}(\varphi_3, \varphi_2)\]
\[\varphi_5 = \text{ZeroExt}(X) \text{ (controlled)}\]
\[\varphi_6 = \text{sub}(\varphi_5, 1)\]
\[\varphi_7 = \text{xor}(\varphi_6, 0x55)\]
\[\varphi_8 = \varphi_7\]
\[\varphi_9 = \text{ptr (constant)}\]
\[\varphi_{10} = \text{offset}\]
\[\varphi_{11} = \text{add}(\varphi_{10}, \varphi_9)\]
\[\varphi_{12} = \text{constant}\]
\[\varphi_{13} = \varphi_{12}\]
\[\varphi_{14} = \text{cmp}(\varphi_8, \varphi_{13})\]
Let's do a DSE on our example

- OK. Now, what the user can control?

**Symbolic Expression Set**

\[
\begin{align*}
φ_1 &= \text{offset (constant)} \\
φ_2 &= \text{SignExt}(φ_1) \\
φ_3 &= \text{ptr (constant)} \\
φ_4 &= \text{add}(φ_3, φ_2) \\
φ_5 &= \text{ZeroExt}(X) \text{ (controlled)} \\
φ_6 &= \text{sub}(φ_5, 1) \\
φ_7 &= \text{xor}(φ_6, 0x55) \\
φ_8 &= φ_7 \\
φ_9 &= \text{ptr (constant)} \\
φ_{10} &= \text{offset} \\
φ_{11} &= \text{add}(φ_{10}, φ_9) \\
φ_{12} &= \text{constant} \\
φ_{13} &= φ_{12} \\
φ_{14} &= \text{cmp}(φ_8, φ_{13})
\end{align*}
\]
Let's do a DSE on our example

• OK. Now, what the user can control?

Symbolic Expression Set

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

φ₁₄ = \text{cmp}(φ₈, φ₁₃)

Formula reconstruction: \text{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)

Formula reconstruction: cmp(φ₇, φ₁₃)
Let's do a DSE on our example

- OK. Now, what the user can control?

**Symbolic Expression Set**

\[ \phi_1 = \text{offset (constant)} \]
\[ \phi_2 = \text{SignExt}(\phi_1) \]
\[ \phi_3 = \text{ptr (constant)} \]
\[ \phi_4 = \text{add}(\phi_3, \phi_2) \]
\[ \phi_5 = \text{ZeroExt}(X) \text{ (controlled)} \]
\[ \phi_6 = \text{sub}(\phi_5, 1) \]
\[ \phi_7 = \text{xor}(\phi_6, 0x55) \]
\[ \phi_8 = \phi_7 \]
\[ \phi_9 = \text{ptr (constant)} \]
\[ \phi_{10} = \text{offset} \]
\[ \phi_{11} = \text{add}(\phi_{10}, \phi_9) \]
\[ \phi_{12} = \text{constant} \]
\[ \phi_{13} = \phi_{12} \]
\[ \phi_{14} = \text{cmp}(\phi_8, \phi_{13}) \]

**Controllable**

**Formula reconstruction:** \( \text{cmp}(\text{xor}(\phi_6, 0x55), \phi_{13}) \)
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

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

\( \phi_1 = \text{offset (constant)} \)
\( \phi_2 = \text{SignExt}(\phi_1) \)
\( \phi_3 = \text{ptr (constant)} \)
\( \phi_4 = \text{add}(\phi_3, \phi_2) \)
\( \phi_5 = \text{ZeroExt}(X) \) (controlled)
\( \phi_6 = \text{sub}(\phi_5, 1) \)
\( \phi_7 = \text{xor}(\phi_6, 0x55) \)
\( \phi_8 = \phi_7 \)
\( \phi_9 = \text{ptr (constant)} \)
\( \phi_{10} = \text{offset} \)
\( \phi_{11} = \text{add}(\phi_{10}, \phi_9) \)
\( \phi_{12} = \text{constant} \)
\( \phi_{13} = \phi_{12} \)

\( \phi_{14} = \text{cmp}(\phi_8, \phi_{13}) \)

Formula reconstruction: \( \text{cmp}(\text{xor}(\text{sub}(\text{ZeroExt}(X), 1), 0x55), \phi_{13}) \)
Let's do a DSE on our example

- OK. Now, what the user can control?

**Symbolic Expression Set**

\( \varphi_1 = \text{offset (constant)} \)
\( \varphi_2 = \text{SignExt}(\varphi_1) \)
\( \varphi_3 = \text{ptr (constant)} \)
\( \varphi_4 = \text{add}(\varphi_3, \varphi_2) \)
\( \varphi_5 = \text{ZeroExt}(X) \) (controlled)
\( \varphi_6 = \text{sub}(\varphi_5, 1) \)
\( \varphi_7 = \text{xor}(\varphi_6, 0x55) \)
\( \varphi_8 = \varphi_7 \)
\( \varphi_9 = \text{ptr (constant)} \)
\( \varphi_{10} = \text{offset} \)
\( \varphi_{11} = \text{add}(\varphi_{10}, \varphi_9) \)
\( \varphi_{12} = \text{constant} \)
\( \varphi_{13} = \varphi_{12} \)
\( \varphi_{14} = \text{cmp}(\varphi_8, \varphi_{13}) \)

**Reconstruction**

Formula reconstruction: \( \text{cmp} \left( \text{xor} \left( \text{sub} \left( \text{ZeroExt}(X), 1 \right), 0x55 \right), \varphi_{12} \right) \)
Let's do a DSE on our example

- OK. Now, what the user can control?

**Symbolic Expression Set**

\[
\begin{align*}
\varphi_1 &= \text{offset (constant)} \\
\varphi_2 &= \text{SignExt}(\varphi_1) \\
\varphi_3 &= \text{ptr (constant)} \\
\varphi_4 &= \text{add}(\varphi_3, \varphi_2) \\
\varphi_5 &= (X) \text{ (controlled)} \\
\varphi_6 &= \text{sub}(\varphi_5, 1) \\
\varphi_7 &= \text{xor}(\varphi_6, 0x55) \\
\varphi_8 &= \varphi_7 \\
\varphi_9 &= \text{ptr (constant)} \\
\varphi_{10} &= \text{offset} \\
\varphi_{11} &= \text{add}(\varphi_{10}, \varphi_9) \\
\varphi_{12} &= \text{constant} \\
\varphi_{13} &= \varphi_{12} \\
\end{align*}
\]

Reconstruction

Formula reconstruction: $\text{cmp}(\text{xor}(\text{sub}(\text{ZeroExt}(X), 1), 0x55), \text{constant})$
Formula reconstruction

- **Formula reconstruction**: \( \text{cmp}(\text{xor}(\text{sub}(	ext{ZeroExt}(X) 1), 0x55), \text{constant}) \)
  - The **constant** is known at runtime: \(0x31\) is the constant for the first iteration

- It is time to use Z3

```python
>>> 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'.

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

\[
\begin{align*}
\text{False} &= ((x - 1) \lor 0x55) \neq 0x31 \\
\text{True} &= ((x - 1) \lor 0x55) = 0x31
\end{align*}
\]

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

$$(((x_1 - 1) \lor 0x55) \neq 0x31)$$
We repeat this operation until all the paths are covered

1. \(((x_1 - 1) \lor 0x55) \neq 0x31)\)

2. \(((x_1 - 1) \lor 0x55) = 0x31) \land ((x_2 - 1) \lor 0x55) \neq 0x3e)\)
We repeat this operation until all the paths are covered

1. $$(((x_1 - 1) \lor 0x55) \neq 0x31)$$

2. $$(((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) \neq 0x3e)$$

3. $$(((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) \neq 0x3d)$$
We repeat this operation until all the paths are covered

1. \(((x_1 - 1) \lor 0x55) \neq 0x31)\n
2. \(((x_1 - 1) \lor 0x55) = 0x31) \land ((x_2 - 1) \lor 0x55) = 0x3e)\n
3. \(((x_1 - 1) \lor 0x55) = 0x31) \land ((x_2 - 1) \lor 0x55) = 0x3e) \land ((x_3 - 1) \lor 0x55) \neq 0x3d)\n
4. \(((x_1 - 1) \lor 0x55) = 0x31) \land ((x_2 - 1) \lor 0x55) = 0x3e) \land ((x_3 - 1) \lor 0x55) = 0x3d) \land ((x_4 - 1) \lor 0x55) \neq 0x26))
We repeat this operation until all the paths are covered

1. \(((x_1 - 1) \lor 0x55) \neq 0x31\)

2. \(((x_1 - 1) \lor 0x55) = 0x31 \land ((x_2 - 1) \lor 0x55) = 0x3e)\)

3. \(((x_1 - 1) \lor 0x55) = 0x31 \land ((x_2 - 1) \lor 0x55) = 0x3e \land ((x_3 - 1) \lor 0x55) \neq 0x3d)\)

4. \(((x_1 - 1) \lor 0x55) = 0x31 \land ((x_2 - 1) \lor 0x55) = 0x3e \land ((x_3 - 1) \lor 0x55) = 0x3d \land ((x_4 - 1) \lor 0x55) \neq 0x26)\)

5. \(((x_1 - 1) \lor 0x55) = 0x31 \land ((x_2 - 1) \lor 0x55) = 0x3e \land ((x_3 - 1) \lor 0x55) = 0x3d \land ((x_4 - 1) \lor 0x55) = 0x26 \land ((x_5 - 1) \lor 0x55) \neq 0x31)\)
We repeat this operation until all the paths are covered

1. \(((x_1 - 1) \lor 0x55) \neq 0x31)\)
2. \(((x_1 - 1) \lor 0x55) == 0x31) \land ((x_2 - 1) \lor 0x55) \neq 0x3e)\)
3. \(((x_1 - 1) \lor 0x55) == 0x31) \land ((x_2 - 1) \lor 0x55) == 0x3e) \land ((x_3 - 1) \lor 0x55) \neq 0x3d)\)
4. \(((x_1 - 1) \lor 0x55) == 0x31) \land ((x_2 - 1) \lor 0x55) == 0x3e) \land ((x_3 - 1) \lor 0x55) == 0x3d) \land ((x_4 - 1) \lor 0x55) == 0x26)\)
5. \(((x_1 - 1) \lor 0x55) == 0x31) \land ((x_2 - 1) \lor 0x55) == 0x3e) \land ((x_3 - 1) \lor 0x55) == 0x3d) \land ((x_4 - 1) \lor 0x55) \neq 0x26) \land ((x_5 - 1) \lor 0x55) \neq 0x31)\)
6. \(((x_1 - 1) \lor 0x55) == 0x31) \land ((x_2 - 1) \lor 0x55) == 0x3e) \land ((x_3 - 1) \lor 0x55) == 0x3d) \land ((x_4 - 1) \lor 0x55) == 0x26) \land ((x_5 - 1) \lor 0x55) == 0x31)\)
The complete formula to return 0 is:

\[ \beta_i = (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) == 0x26) \land (((x_5 - 1) \lor 0x55) == 0x31)) \]

Where \( x_1, x_2, x_3, x_4 \) and \( x_5 \) are five variables controlled by the user inputs.

The complete formula to return 1 is:

\[ \beta_{i+1} = (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) == 0x26) \land (((x_5 - 1) \lor 0x55) != 0x31) \lor (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) != 0x3d) \lor (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) != 0x3d) \lor (((x_2 - 1) \lor 0x55) == 0x3e) \lor (((x_1 - 1) \lor 0x55) != 0x31)) \]

Where \( x_1, x_2, x_3, x_4 \) and \( x_5 \) are five variables controlled by the user inputs.
Generate a concrete Value to return 0

- The complete formula to return 0 is:
  \[
  \beta_i = (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) == 0x26) \land (((x_5 - 1) \lor 0x55) == 0x31))
  \]

- The concrete value generation using z3

```python
>>> 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)
elite
```
The complete formula to return 1 is:

$$\beta^{(i+1)} = (((x_1 - 1) \land 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) == 0x26) \land (((x_5 - 1) \lor 0x55) != 0x31) \lor (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) != 0x26) \lor (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) != 0x26) \lor (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) != 0x26) \lor (((x_1 - 1) \lor 0x55) == 0x31) \land (((x_2 - 1) \lor 0x55) == 0x3e) \land (((x_3 - 1) \lor 0x55) == 0x3d) \land (((x_4 - 1) \lor 0x55) != 0x26) \lor (((x_1 - 1) \lor 0x55) == 0x31)$$

The concrete value generation using z3

```python
>>> 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) == 0x3d), (((x3 - 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
- $\beta$ represents a symbolic path expression
- To cover the function $\text{check\_password}$ we must generate a concrete value for each $\beta$ 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 $\beta$. 
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
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 Bédrule and Aurélien Wailly for their feedbacks.

- **Social event**
  - Don't forget the doar-e social event after the talks, there are some free beers!