

The Reverse Engineering Language REIL and its applications

Sebastian Porst (sebastian.porst@zynamics.com)
zynamics GmbH

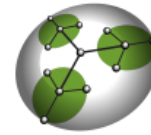


HITB SECCONF 2009

DU دبي BAI

20TH - 23RD APRIL 2009

I write code for



... sometimes I **debug** other people's code

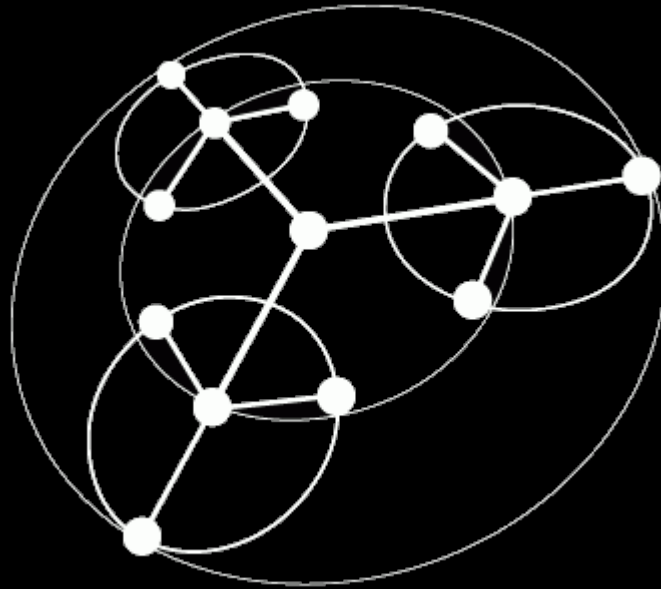
A screenshot of assembly code from a debugger. The top section shows instructions at addresses 7C8346C1, 7C8346C3, and 7C8346C9. A green arrow points from the instruction at 7C8346C3 to a block of code starting at 7C834675. The code at 7C834675 includes instructions for kernel32.dll: ::_ParseNumber@32, and increments and jumps. The instruction at 7C834675 is highlighted in green.

```
7C8346C1 test     ecx, ecx
7C8346C3 mov     ss:[ebp+var_AE], ecx
7C8346C9 jnz     cs:loc_7C84CD8E

7C834675 kernel32.dll: ::_ParseNumber@32
7C83467E inc     eax
7C834680 inc     eax
7C834682 jmp     cs:loc_7C8346CF
```

We love graphs

We love static analysis



We want to improve binary code RE

REIL

Reverse Engineering Intermediate Language

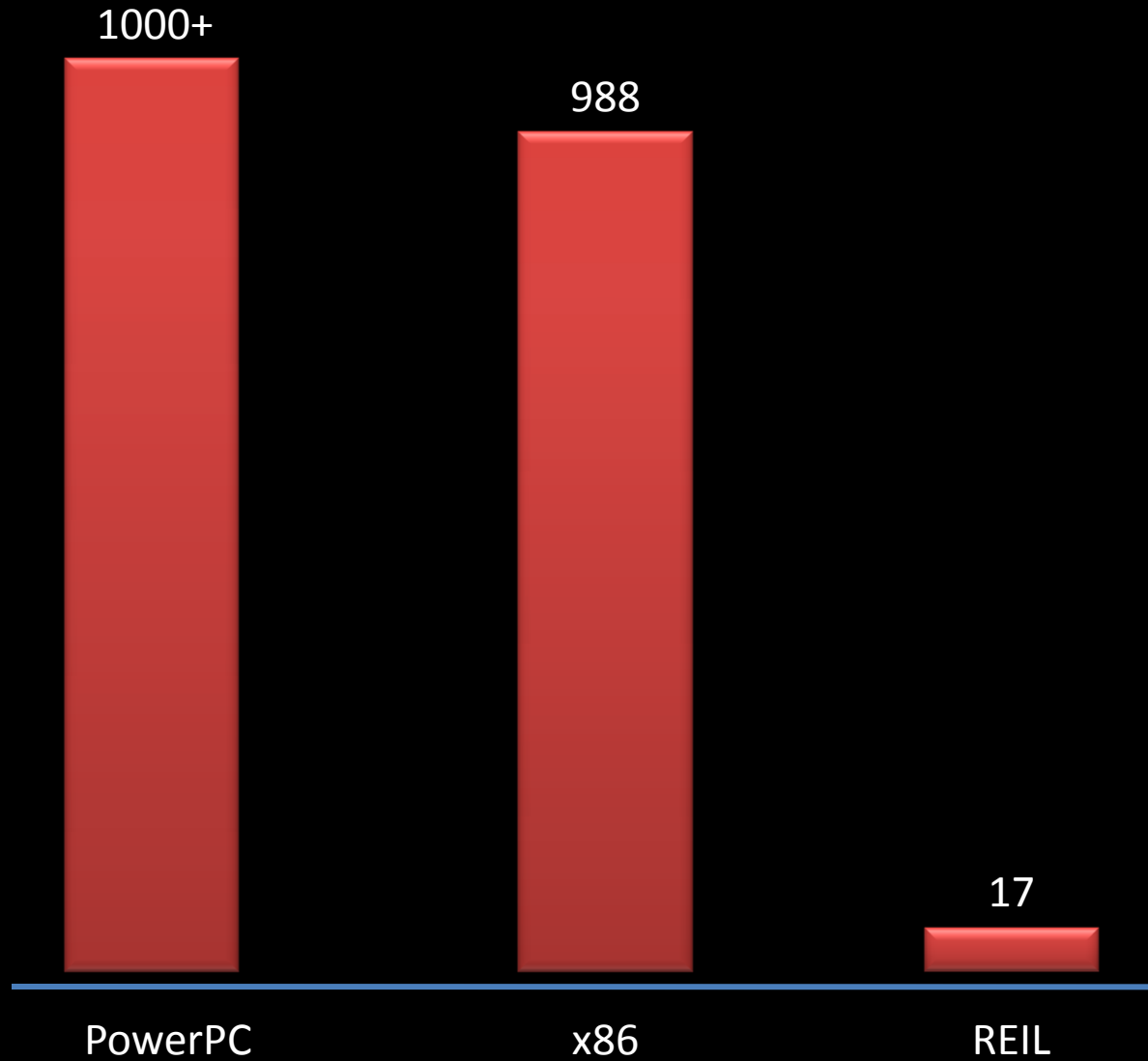
Three Good Reasons for REIL

Simple

Free of side-effects

Platform-Independent

Fewer Instructions



X86 Code

PowerPC Code

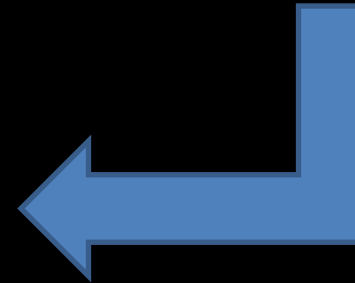
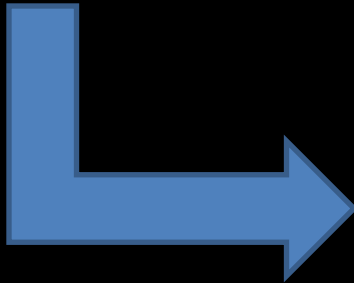
ARM Code



X86 Translator

PowerPC Translator

ARM Translator



REIL Code



MonoREIL

Other Analysis
Algorithms

The diagram is shaped like a house. The roof is a red triangle containing the text 'REIL VM Architecture'. Below the roof is a large black rectangle with a white border containing the text 'Register-based Virtual Machine', 'Infinite number of registers', and 'Infinite amount of memory'. At the bottom is a smaller black rectangle with a white border containing the text 'Close to the original architecture'.

REIL VM Architecture

Register-based Virtual Machine

Infinite number of registers

Infinite amount of memory


Close to the original architecture

17

Instructions

add
and
bisz
bsh
div
jcc
ldm
mod
mul
nop
or
stm
str
sub
undef
unkn
xor

40131A.07 add (t0, b4), (t1, b4), (t2, b8), [(foo, bar)]



1 REIL Address

Native Part

REIL Part

1 REIL Mnemonic

3 Operands

∞ Metadata

1 REIL Operand

2 (really 3) Pieces of Information

3 Operand Types

7 Operand Sizes

∞ Operand Values

Six Arithmetic Instructions

Addition

Logical Shift

Unsigned Division

Unsigned Modulo

Unsigned Multiplication

Subtraction

add

and

bisz

bsh

div

jcc

ldm

mod

mul

nop

or

stm

str

sub

undef

unkn

xor

Three Bitwise Instructions

Bitwise AND

Bitwise OR

Bitwise XOR

add

and

bisz

bsh

div

jcc

ldm

mod

mul

nop

or

stm

str

sub

undef

unkn

xor

Three Data Transfer Instructions

Load Memory

Store Memory

Transfer Register Content

add

and

bisz

bsh

div

jcc

ldm

mod

mul

nop

or

stm

str

sub

undef

unkn

xor

Two **Logical** Instructions

Compare to Zero

Jump Conditional

add

and

bisz

bsh

div

jcc

ldm

mod

mul

nop

or

stm

str

sub

undef

unkn

xor

Three **Other** Instructions

No Operation

Undefine Register

Unknown Mnemonic

add

and

bisz

bsh

div

jcc

ldm

mod

mul

nop

or

stm

str

sub

undef

unkn

xor

All Instructions are equal ...

```
add t0, t1, t2
and t0, t1, t2
bsh t0, t1, t2
div t0, t1, t2
mod t0, t1, t2
mul t0, t1, t2
or  t0, t1, t2
sub t0, t1, t2
xor t0, t1, t2
```

... but some instructions are
more equal than others

```
bisz t0, , t2
jcc  t0, , t2
ldm  t0, , t2
nop  , ,
stm  t0, , t2
str  t0, , t2
undef , , t2
unkn , ,
```


What REIL can't do

FPU Instructions `fadd`
`fabbs`
`fdivp`
`frsp.`
`fnmsubs`
...

System Instructions `int`
`sidt`
`bkpt`
`rfi`
`sc`
...

Weird Instructions `setend`
`pld`
`mcrr`
...

THE FUTURE!

More Architectures (x64, MIPS, ...)

More Operand Information

More Instructions

So ...

... what is it all good for?

Register Tracking

Follow the **effects**
of a **register**
on the **program state**

Demo

Register Tracking

Negative Array Access

MS08-67

Demo

MS08-67

... but aren't those really difficult to implement?

NO!



MonoREIL

Create an Instruction Graph

```
graph TD; A[Create an Instruction Graph] --> B[Define a Lattice]; B --> C[Define Transformations]; C --> D[Define a Graph Walker]; D --> E[Define a Start Vector];
```

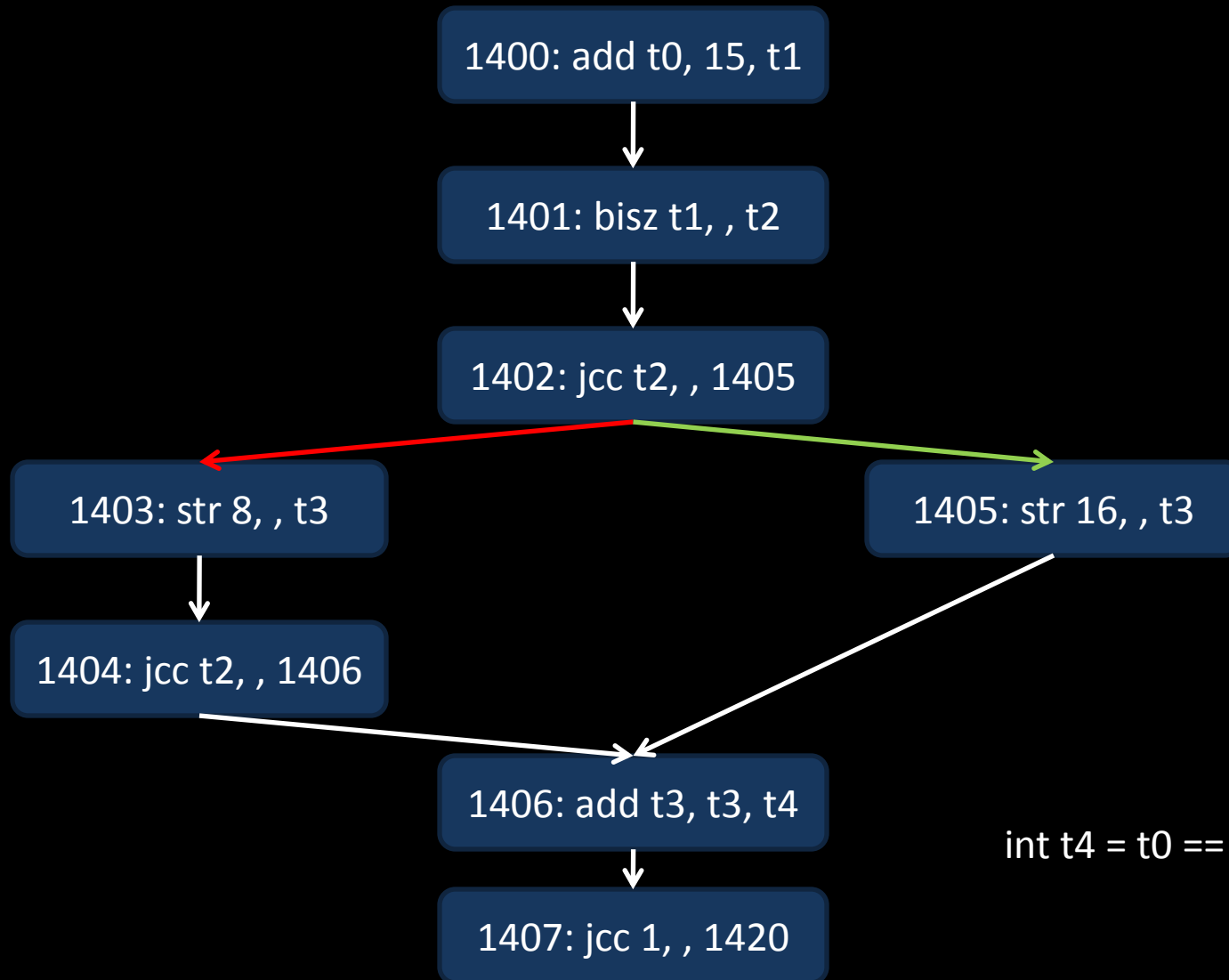
Define a Lattice

Define Transformations

Define a Graph Walker

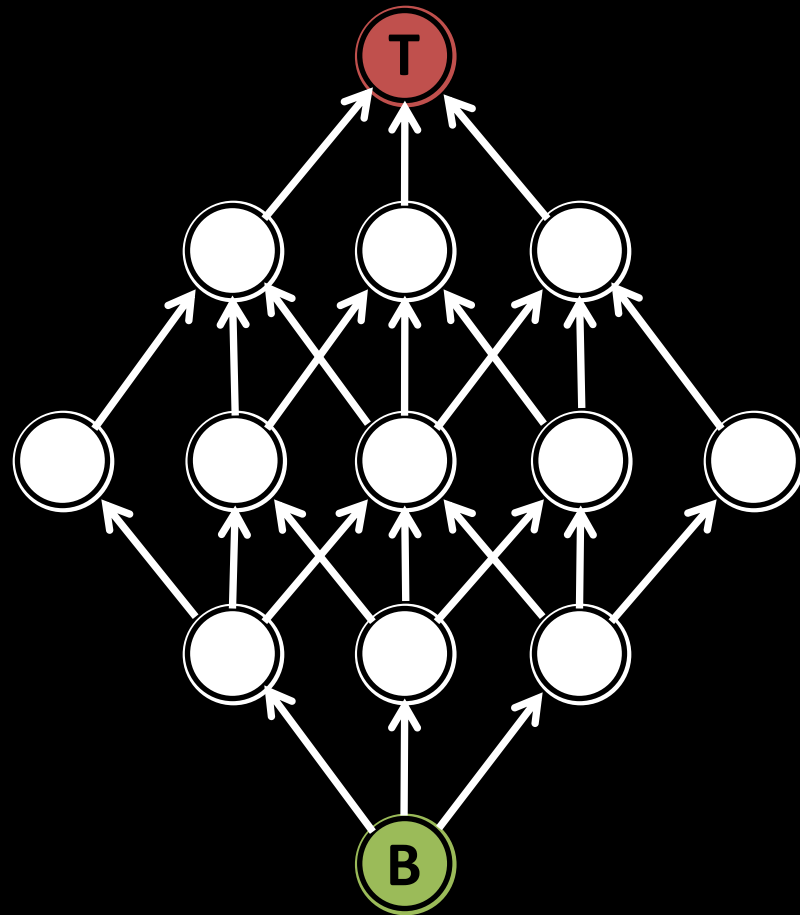
Define a Start Vector

Create an Instruction Graph

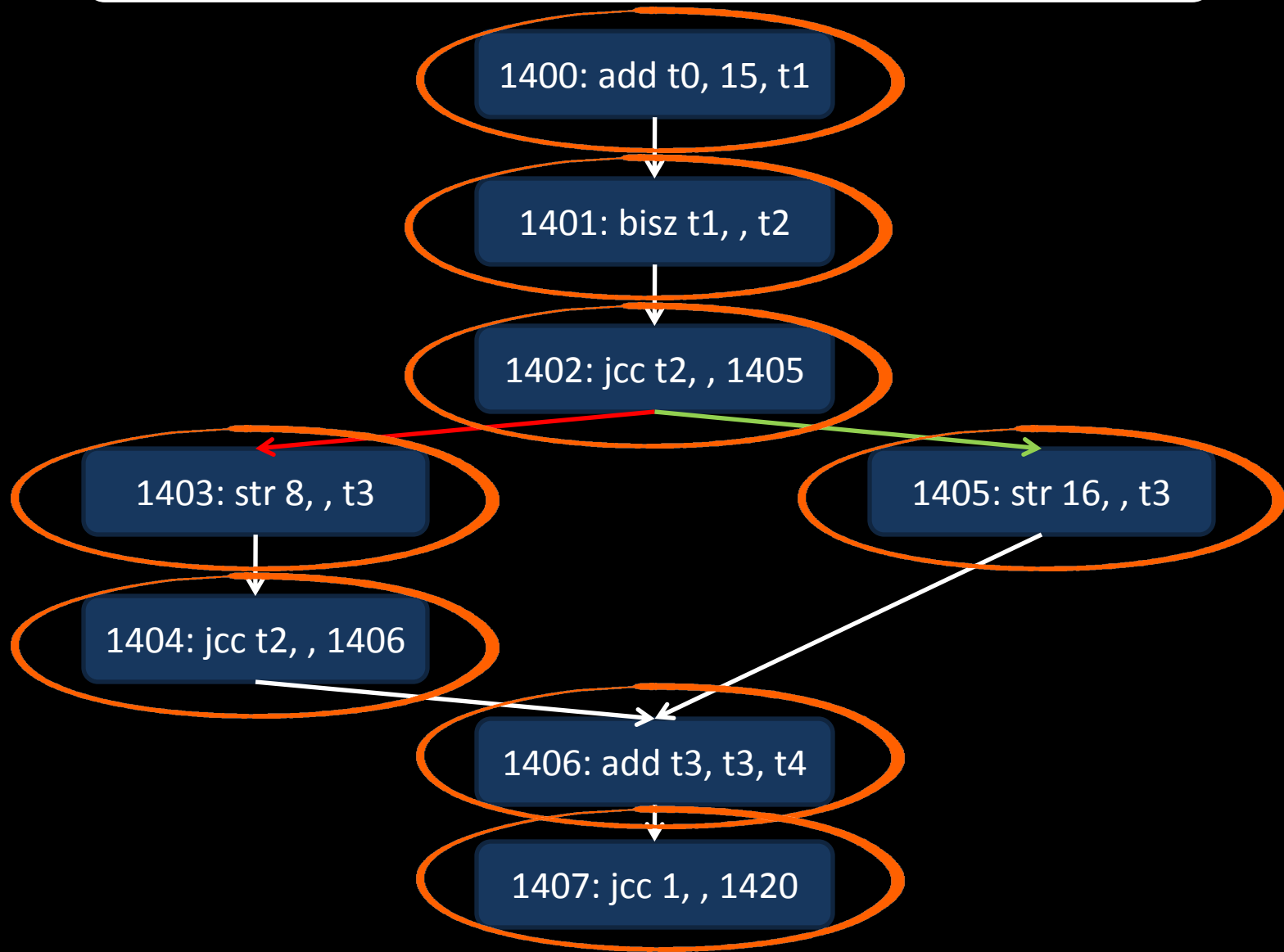


int t4 = t0 == -15 ? 32 : 16

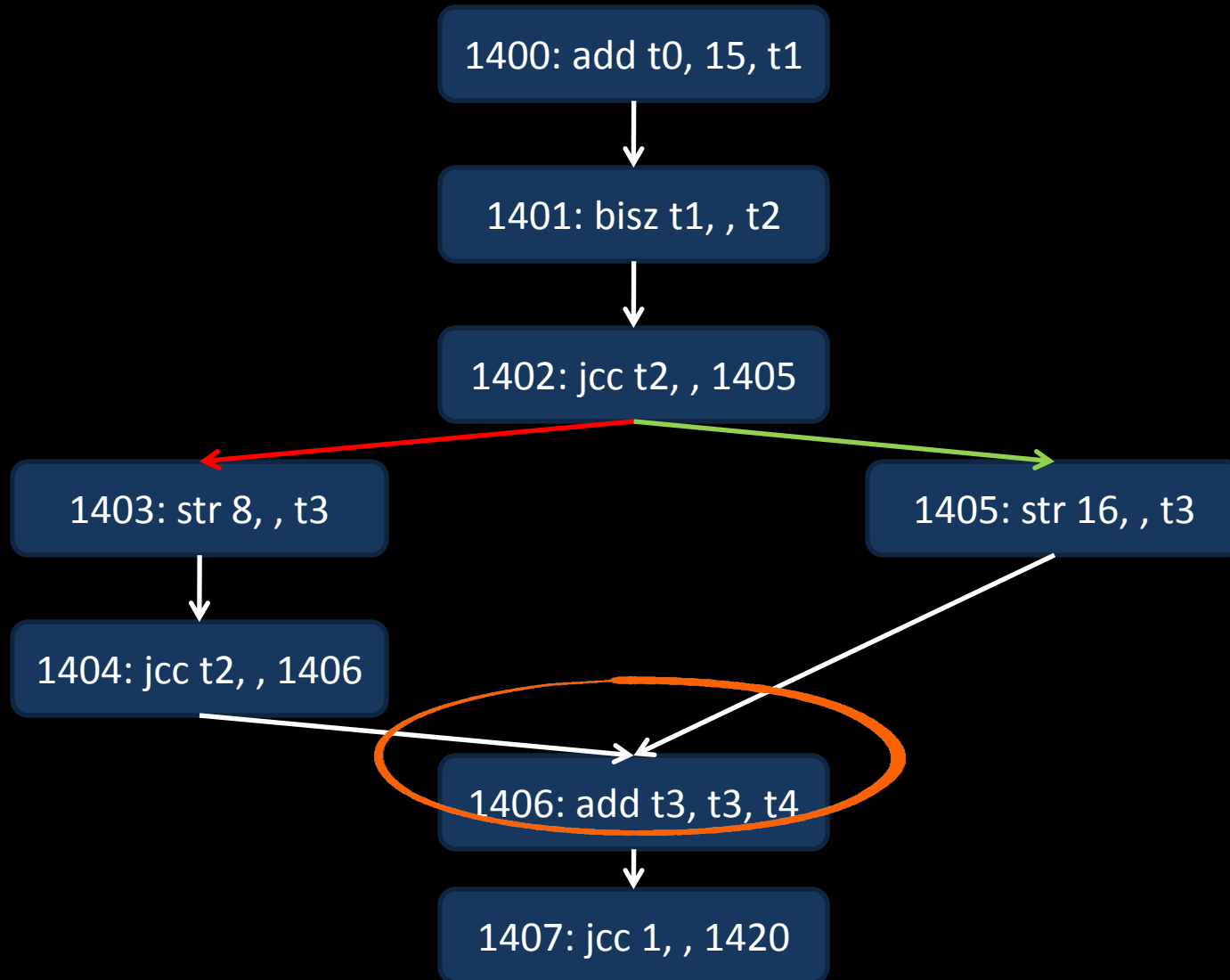
Define Lattice Elements



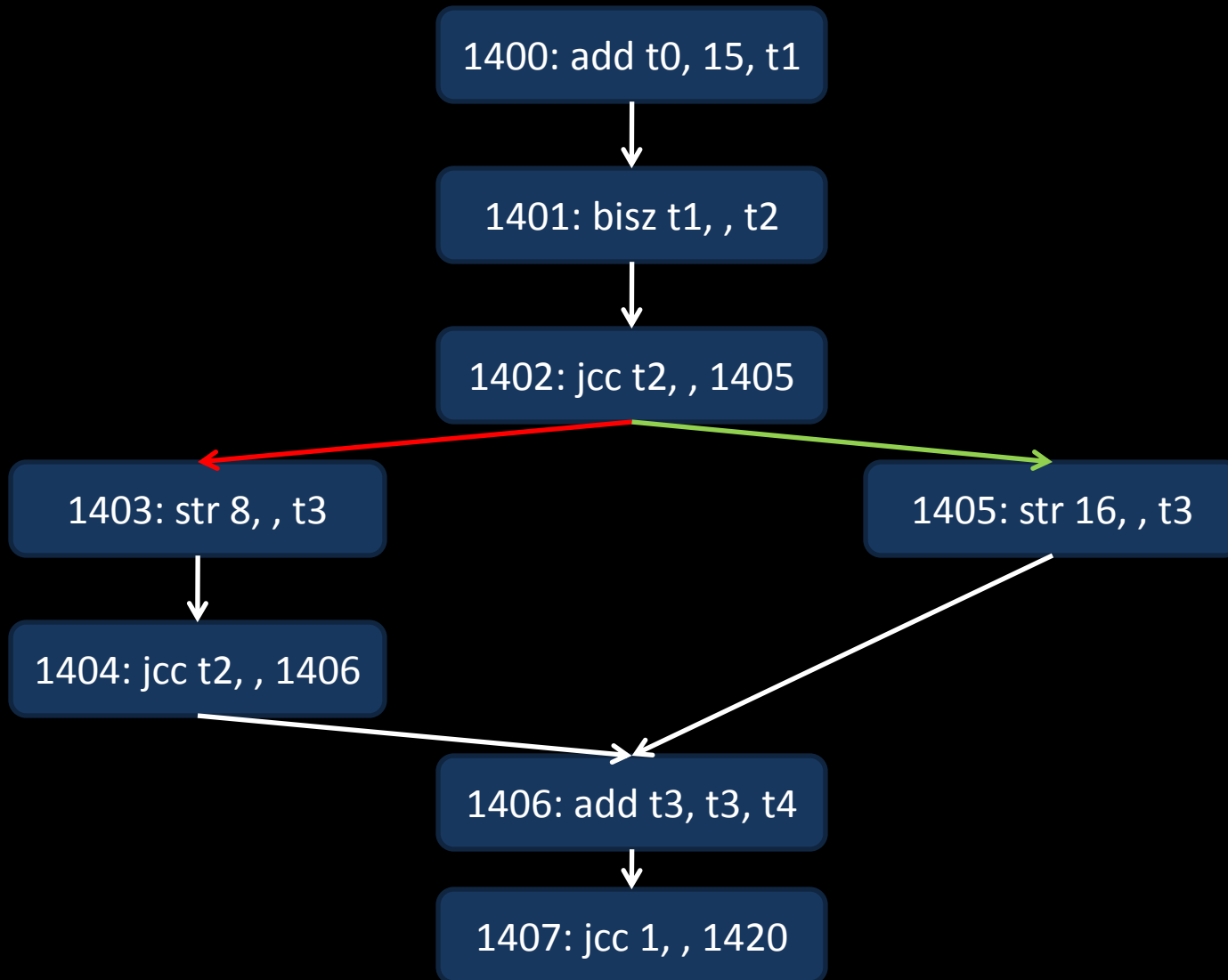
Define Transformations



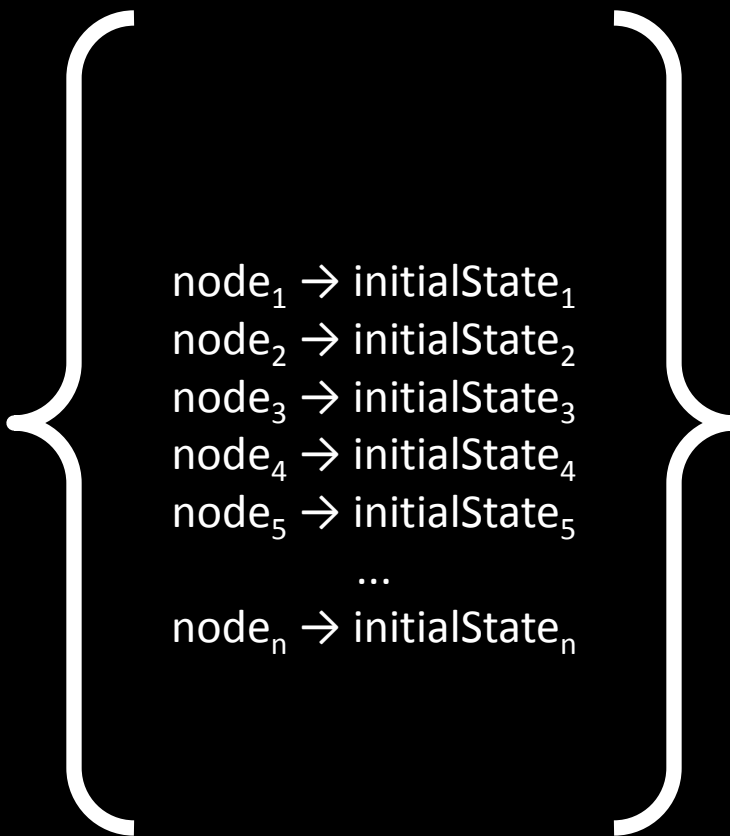
Define a Join function



Define a Graph Walker

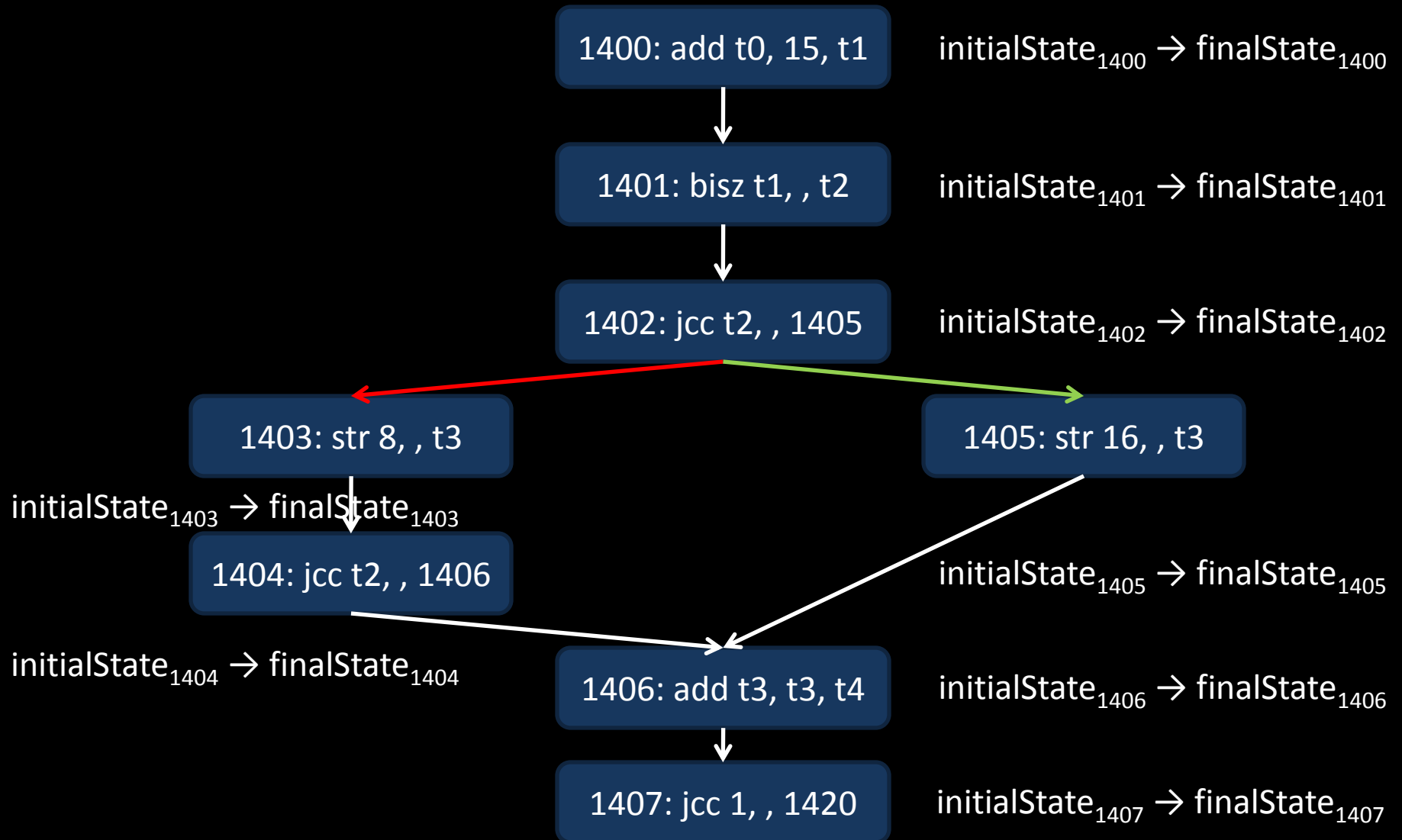


Define a Start Vector

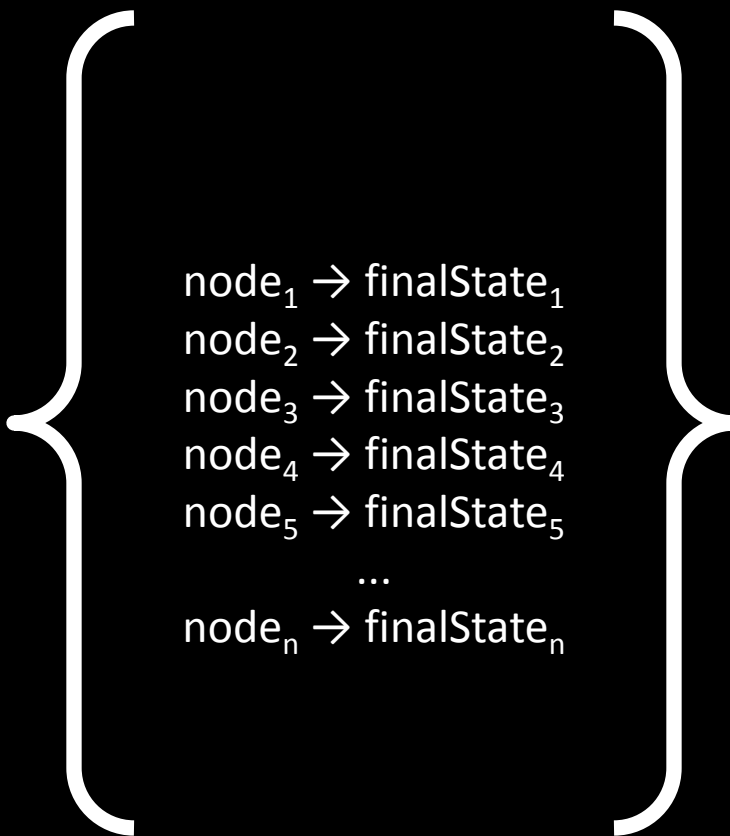


node₁ → initialState₁
node₂ → initialState₂
node₃ → initialState₃
node₄ → initialState₄
node₅ → initialState₅
...
node_n → initialState_n

Running MonoREIL

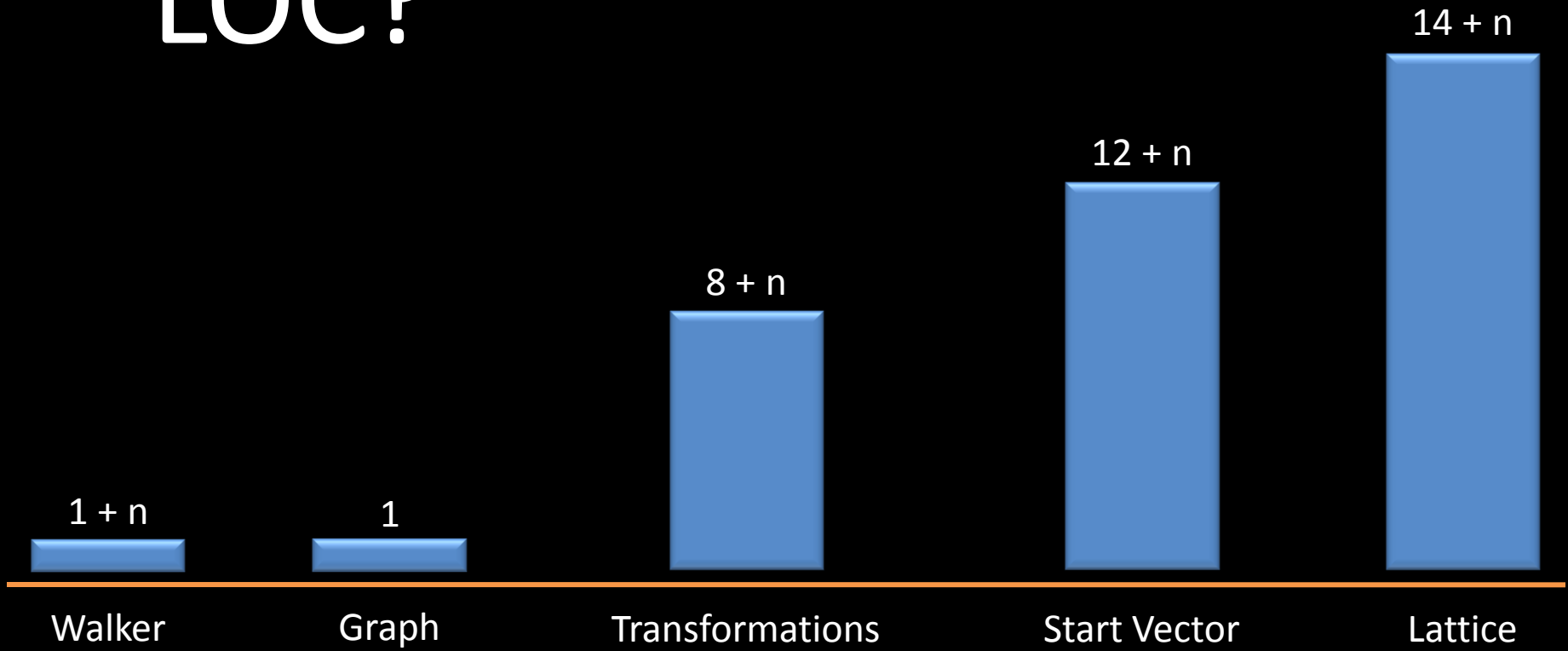


The Result

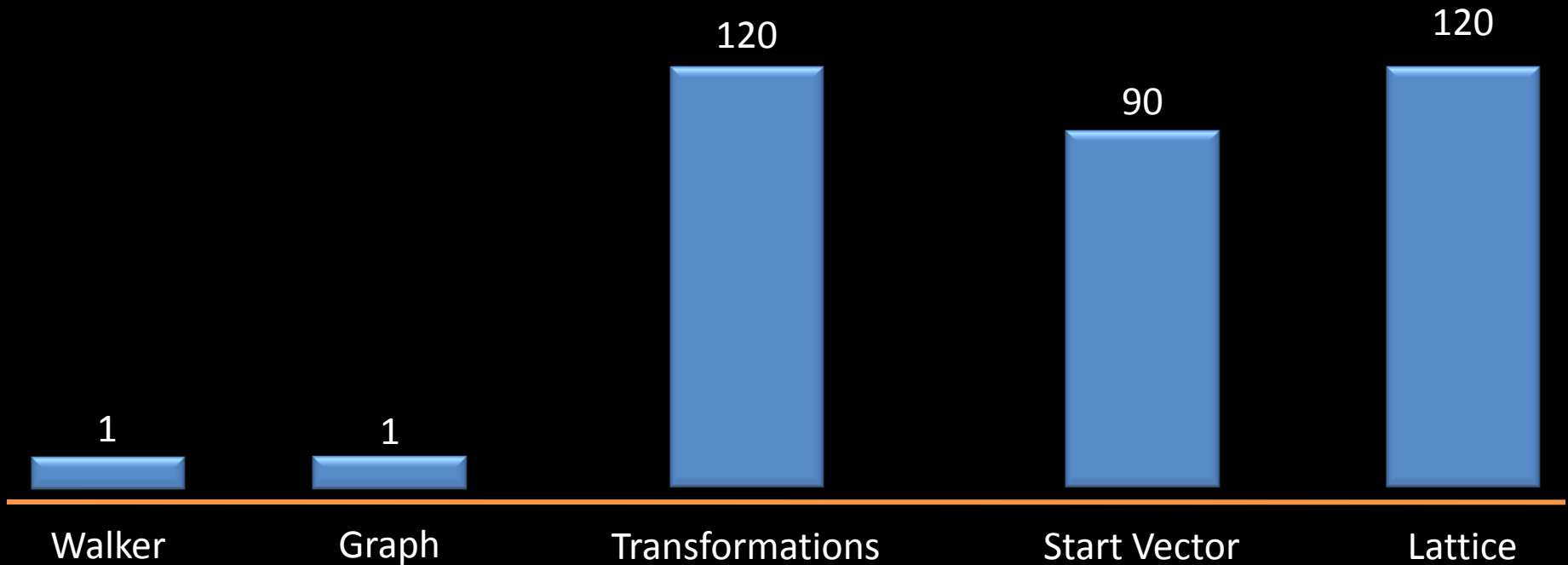


node₁ → finalState₁
node₂ → finalState₂
node₃ → finalState₃
node₄ → finalState₄
node₅ → finalState₅
...
node_n → finalState_n

LOC?



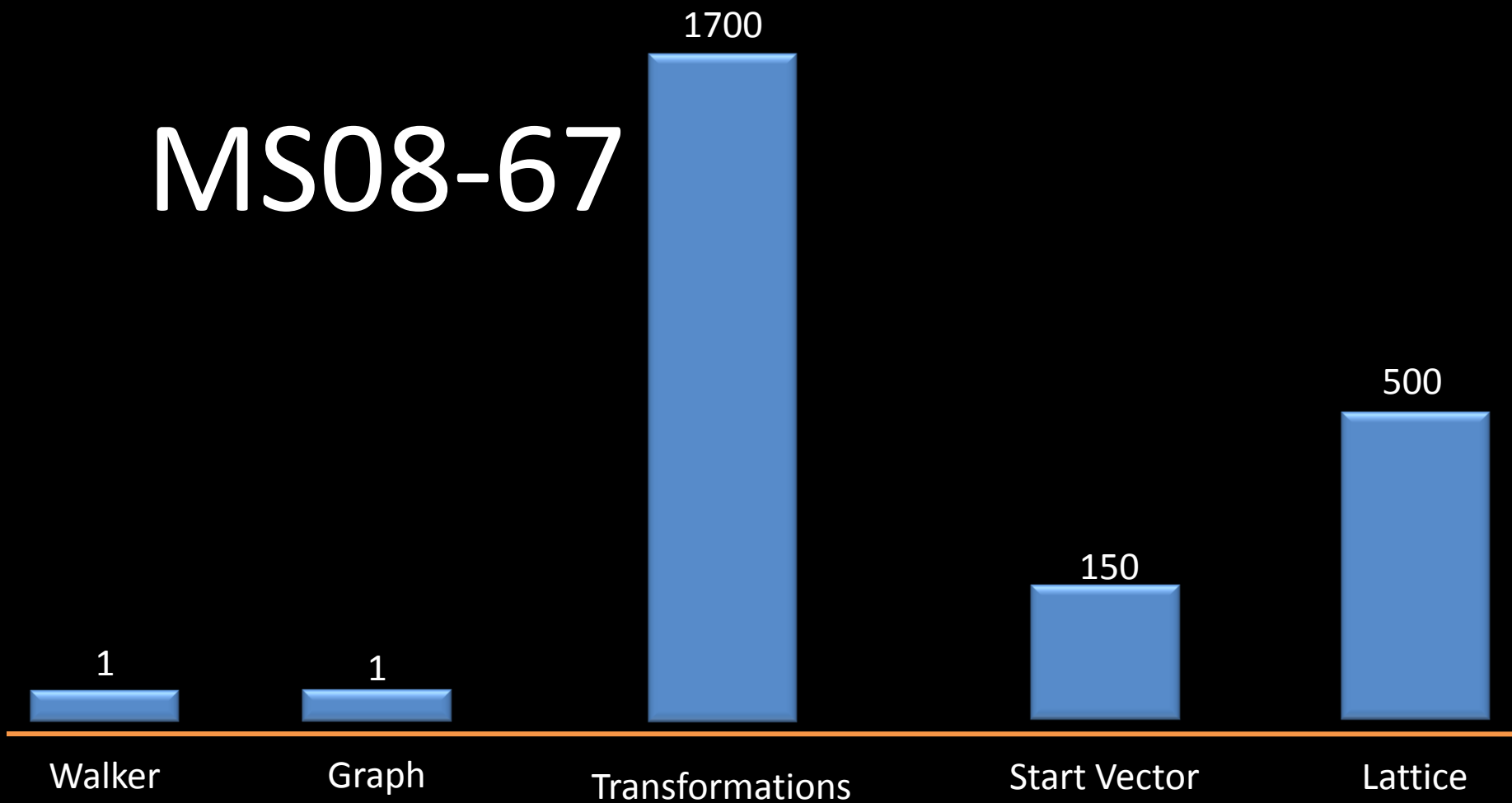
Register Tracking



Demo

Register Tracking Code

MS08-67



Demo

MS08-67

THE FUTURE!

Deobfuscating Optimizer

Type Reconstruction

Related Work

ERESI

Vine (BitBlaze)

Silvio Cesare

Hex-Rays

Thank You!

Questions?