SimuVEX

Using VEX in Symbolic Analysis

seclab
THE COMPUTER SECURITY GROUP AT UC SANTA BARBARA

Yan Shoshitaishvili
yans@cs.ucsb.edu
2014

Who am I?

My name is Yan Shoshitaishvili, and I am a PhD student in the Seclab at UC Santa Barbara.

Email: yans@cs.ucsb.edu

Twitter: @Zardus

Github: http://github.com/zardus

Blog: http://blog.yancomm.net

This work is a collaboration between the UCSB Seclab and the Northeastern Seclab!

Don't Panic!

This presentation **does** have a design!

1. Who (are we)?
2. What (is Symbolic Analysis)?
3. Why (did we choose VEX)?
4. How (do we do it)?
5. Where (does all of this get us)?
6. When (will it be released)?

Why Symbolic Analysis?

"How do I trigger path X or condition Y?"

- Dynamic analysis
  - Input A? No. Input B? No. Input C? …
  - Based on concrete inputs to application.
- (Concrete) static analysis
  - "You can't"/"You might be able to"
  - Based on various static techniques.

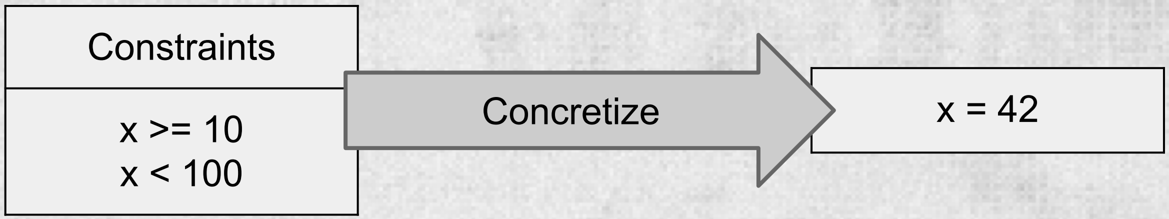
We need something slightly different.

What is Symbolic Analysis?

"How do I trigger path X or condition Y?"

1. Interpret the application.
2. Track "constraints" on variables.
3. When the required condition is triggered, "concretize" to obtain a possible input.

"Concretize"?



Constraint solving:

- ❏ Conversion from set of constraints to set of concrete values that satisfy them.
- ❏ NP-complete, in general.

Symbolic Execution Example

```
x = int(input())
if x >= 10:
  if x < 100:
    print "Two!"
  else:
    print "Lots!"
else:
  print "One!"
```

Symbolic Execution Example

```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```

State A

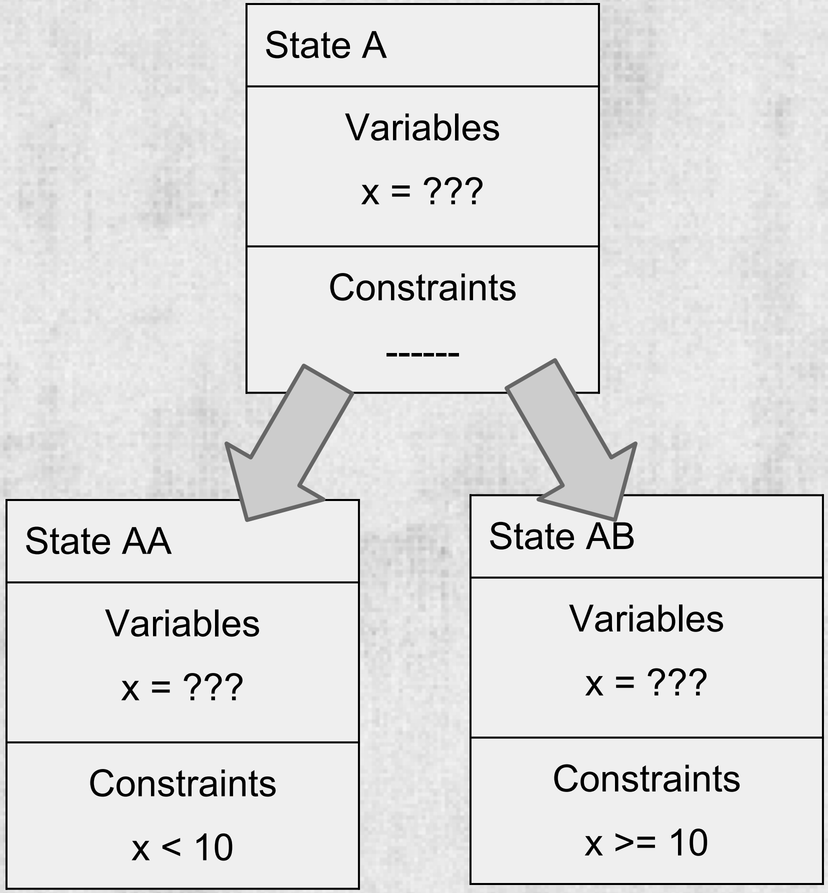
Variables

x = ???

Constraints

Symbolic Execution Example

```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```



State A

Variables

x = ???

Constraints

State AA

Variables

x = ???

Constraints

x < 10

State AB

Variables

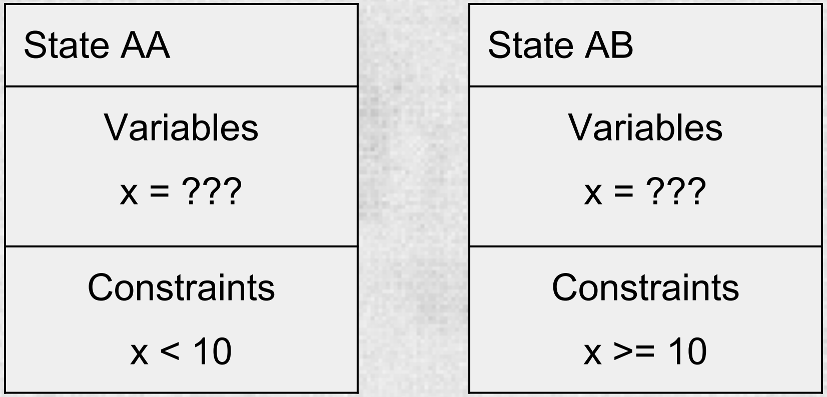
x = ???

Constraints

x >= 10

Symbolic Execution Example

```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```



State AA

Variables

x = ???

Constraints

x < 10

State AB

Variables

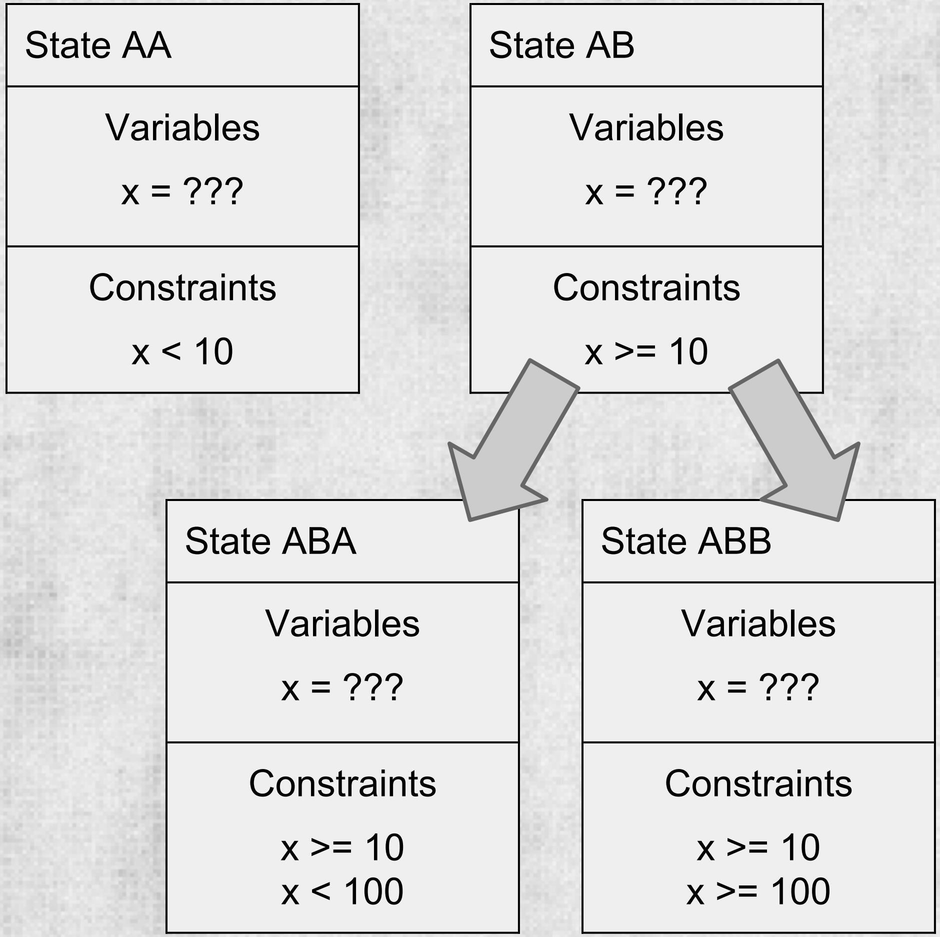
x = ???

Constraints

x >= 10

Symbolic Execution Example

```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```



State AA

Variables

x = ???

Constraints

x < 10

State AB

Variables

x = ???

Constraints

x >= 10

State ABA

Variables

x = ???

Constraints

x >= 10
x < 100

State ABB

Variables

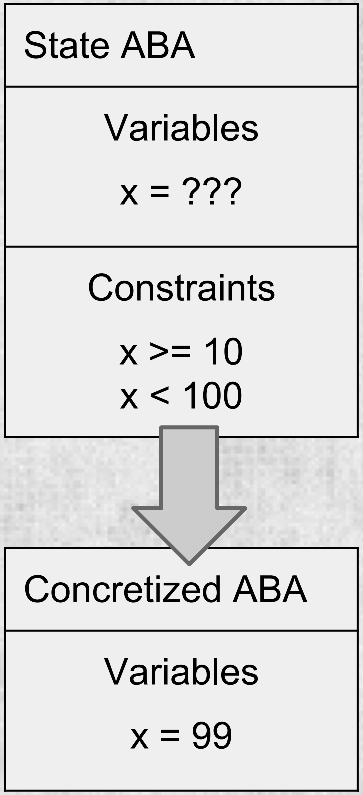
x = ???

Constraints

x >= 10
x >= 100

Concretization Time!

```
x = int(input())
if x >= 10:
    if x < 100:
        print "Two!"
    else:
        print "Lots!"
else:
    print "One!"
```



State ABA

Variables

x = ???

Constraints

x >= 10
x < 100

Concretized ABA

Variables

x = 99

Symbolic Analysis Is Useful

Lots of uses:

- ❏ Reasoning about reachability
- ❏ Bughunting
- ❏ Test-case generation

Symbolic Analysis Is Hard

Two main challenges unique to symbolic analysis:

1. Constraint Solving
   a. NP-complete, in general
   b. "not our field"
2. State Explosion
   a. All outcomes of a piece of code must be considered.
   b. Loops!

Reinventing the Wheel

Existing systems:

1. Source level: EXE, CUTE, **KLEE**, AEG
2. Binary level: Mayhem, **Fuzzball, Avalanche**
3. System level: **S2E**

Hard to find a balance of flexibility, usability, and support.

Stand on the Shoulders of Giants

Balance between fine-grained control and existing tool/idea reuse:

Concepts: related work

Binary translation: VEX

Constraint solving: Z3

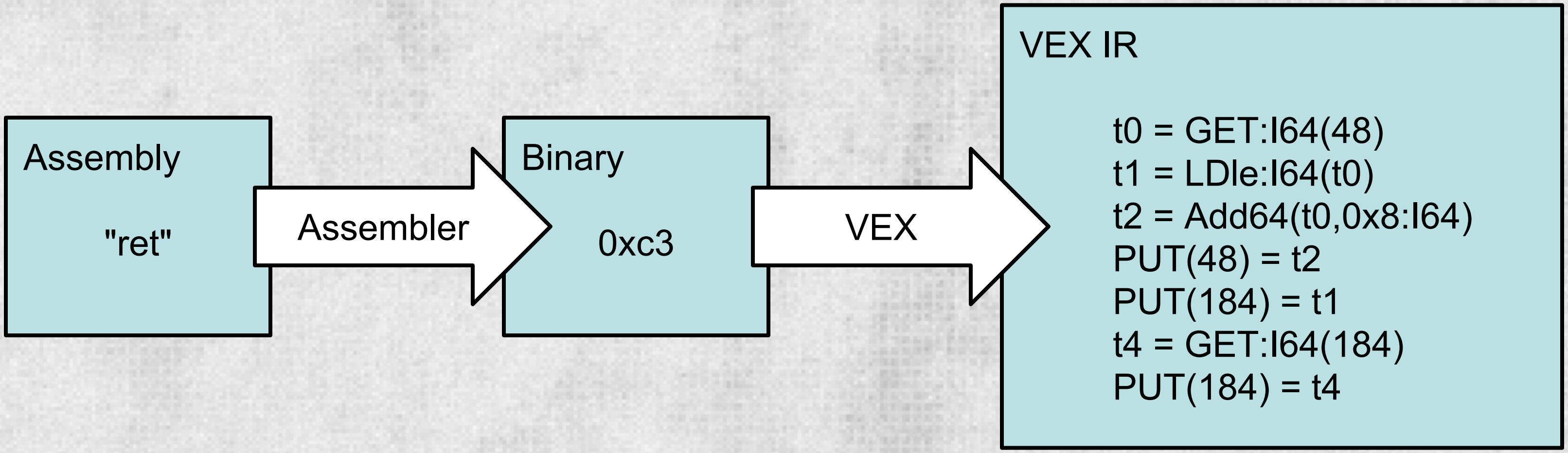
Why Z3?

"Shared-source" constraint solver from Microsoft Research.

❑ Actively developed
❑ Powerful and flexible
❑ Python bindings!
❑ Not too hard to switch away from!

VEX Crash Course

VEX is Valgrind's intermediate language, allowing Valgrind's tools to be implemented once for cross-platform analyses.



Assembly

"ret"

Assembler

Binary

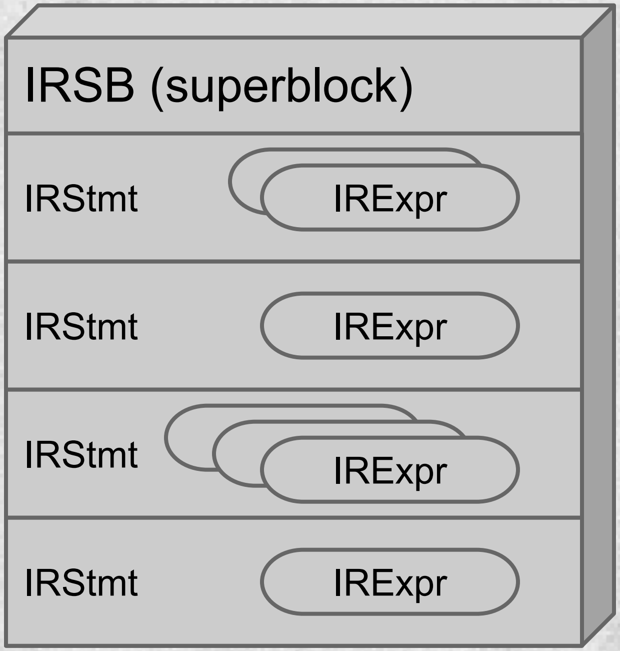
0xc3

VEX

VEX IR

```
t0 = GET:I64(48)
t1 = LDle:I64(t0)
t2 = Add64(t0,0x8:I64)
PUT(48) = t2
PUT(184) = t1
t4 = GET:I64(184)
PUT(184) = t4
```

Code VEXonomy

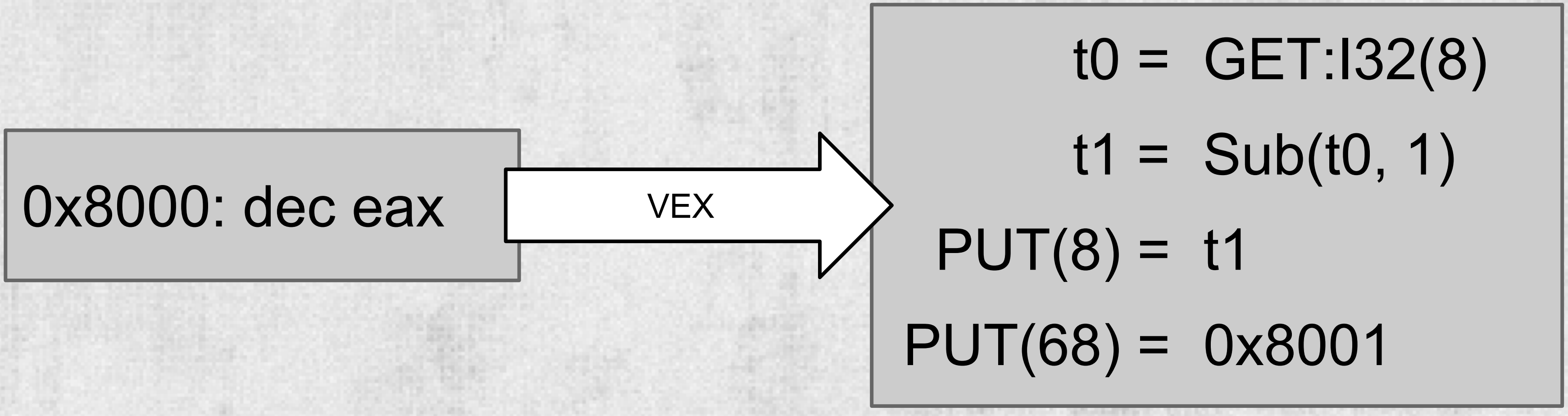


VEX translates instructions to IRExprs, IRStmts, IRSBs.

- ❑ IRExprs provide the values
- ❑ IRStmts "describe" state changes
- ❑ IRSBs maintain structure/order

Creates a reproducible, side-effects-free representation.

Step-by-step VEXample



0x8000: dec eax

```
      t0 =  GET:I32(8)
      t1 =  Sub(t0, 1)
  PUT(8) =  t1
 PUT(68) =  0x8001
```

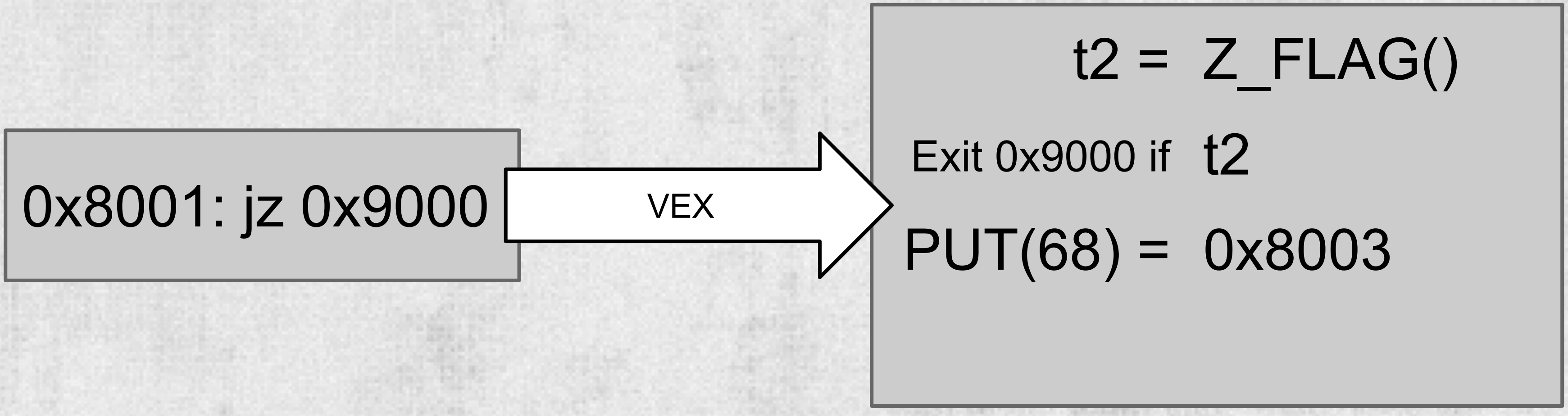
IRStmt: set t0 to... IRExpr: value of eax

IRStmt: set t1 to... IRExpr: t0 - 1

IRStmt: put into eax... IRExpr: t1

IRStmt: put into eip... IRExpr: addr of next instruction

Step-by-step VEXample (2)



0x8001: jz 0x9000

VEX

t2 = Z_FLAG()

Exit 0x9000 if t2

PUT(68) = 0x8003

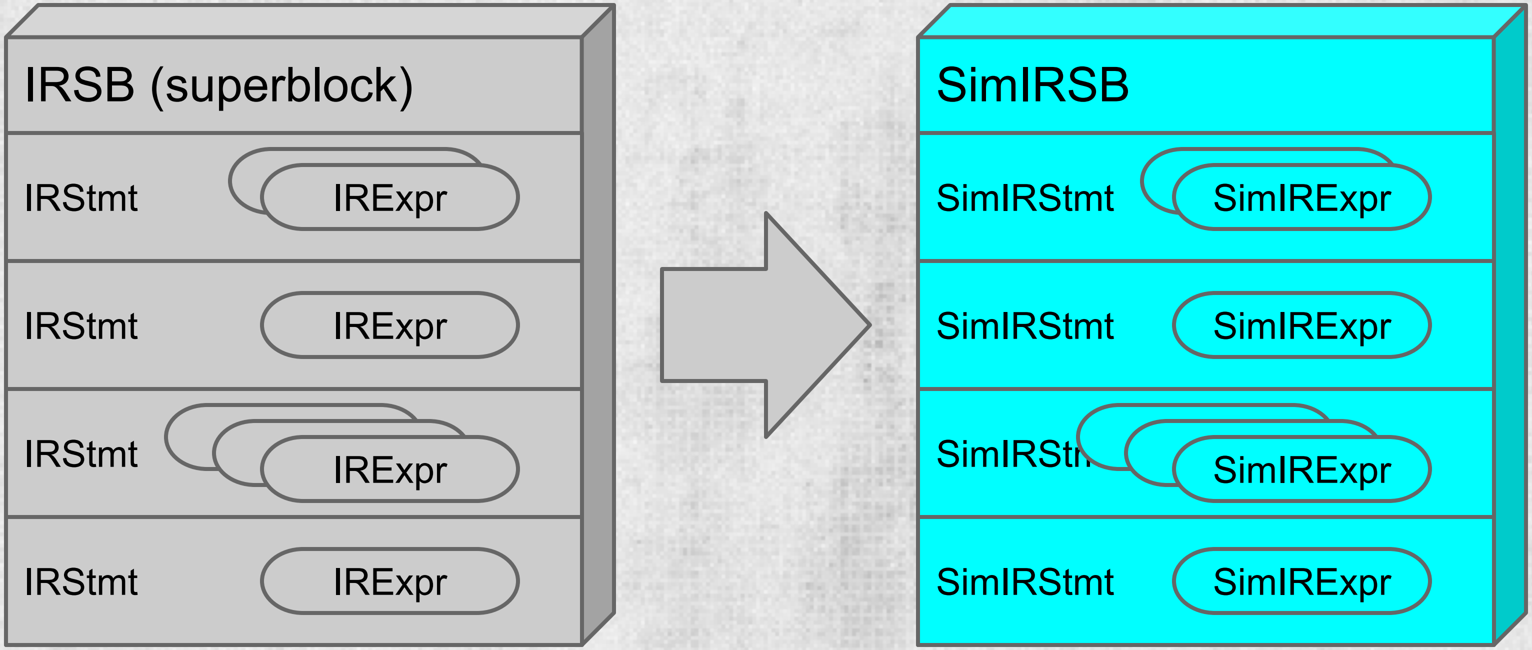
IRStmt: set t0 to... IRExpr: value of eax

IRStmt: exit to 0x9000 if... IRExpr: t0

IRStmt: put into eip... IRExpr: addr of next instruction

VEXamorphosis

SimuVEX creates a symbolic interpretation layer over VEX:



VEXterpretation

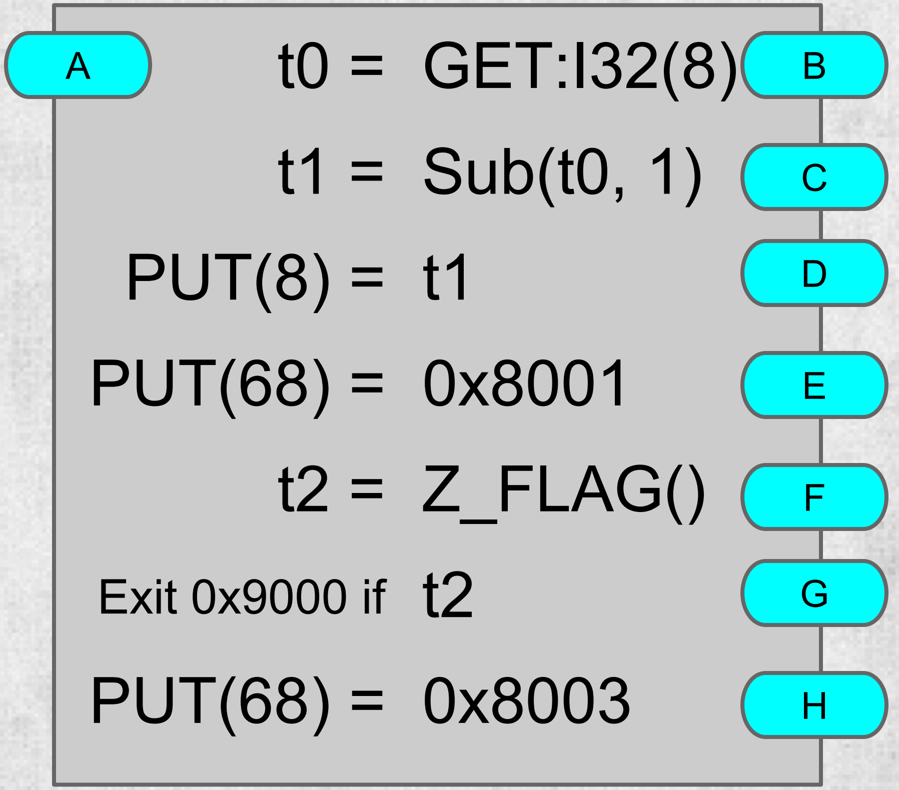
- SimIRExprs represent symbolic values.
- SimIRStmts modify a symbolic state.

What's a symbolic state?

SimState

- symbolic memory
- symbolic registers
- constraints
- plugins
  - (symbolic) 'kernel' state for userspace binaries

VEXterpretation Example



```
          t0 =  GET:I32(8)
          t1 =  Sub(t0, 1)
      PUT(8) =  t1
     PUT(68) =  0x8001
          t2 =  Z_FLAG()
Exit 0x9000 if  t2
     PUT(68) =  0x8003
```

State H

Variables

eax_0

Temps

t0 = eax_0
t1 = eax_0 - 1
t2 = eax_0-1 == 0

Registers

eax = eax_0 - 1
eip = 0x8003

Constraints

eax_0 - 1 != 0

State G1

Variables

eax_0

Temps

t0 = eax_0
t1 = eax_0 - 1
t2 = eax_0-1 == 0

Registers

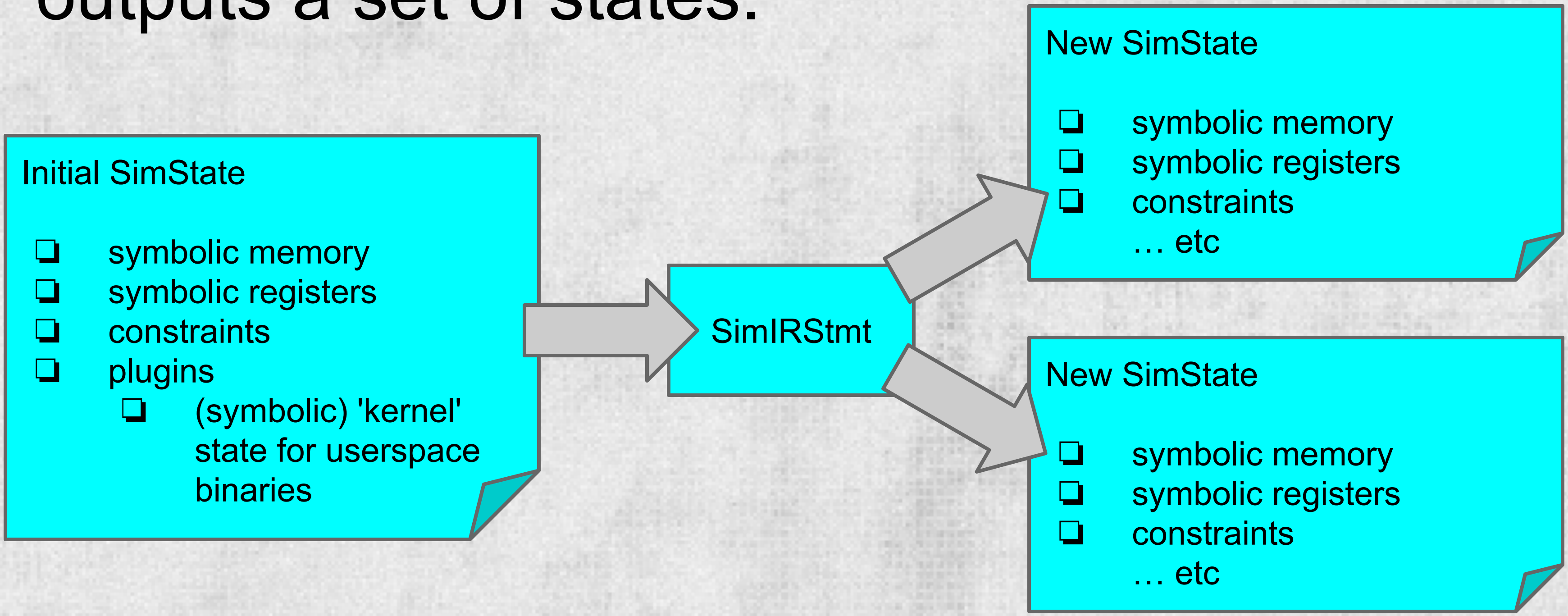
eax = eax_0 - 1
eip = 0x9000

Constraints

eax_0 - 1 == 0

Symbolic Interpretation (IRStmt)

Every SimIRStmt takes a state, makes changes to memory, registers, and constraints, and outputs a set of states.



Initial SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints
- ❑ plugins
  - ❑ (symbolic) 'kernel' state for userspace binaries

SimIRStmt

New SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints

… etc

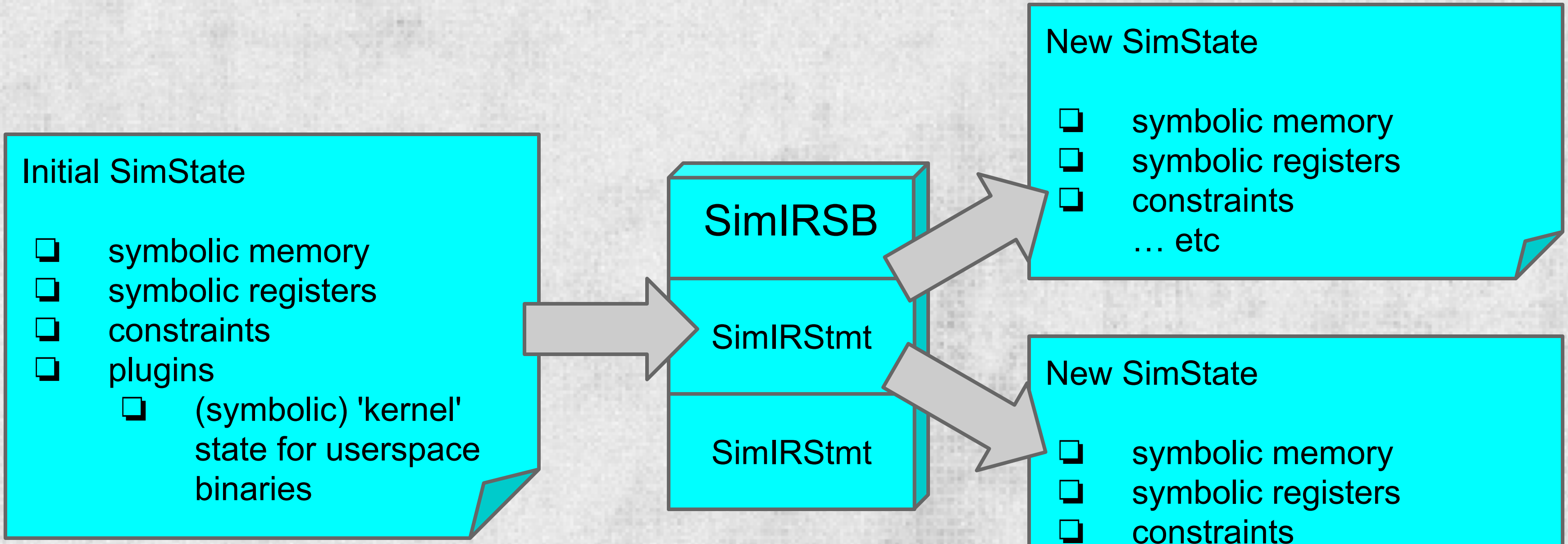
New SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints

… etc

Symbolic Interpretation (IRSB)

These statements are aggregated in SimIRSBs.



Initial SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints
- ❑ plugins
  - ❑ (symbolic) 'kernel' state for userspace binaries

SimIRSB

SimIRStmt

SimIRStmt

New SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints

… etc

New SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints

… etc

Complications...

The naive approach has some issues.

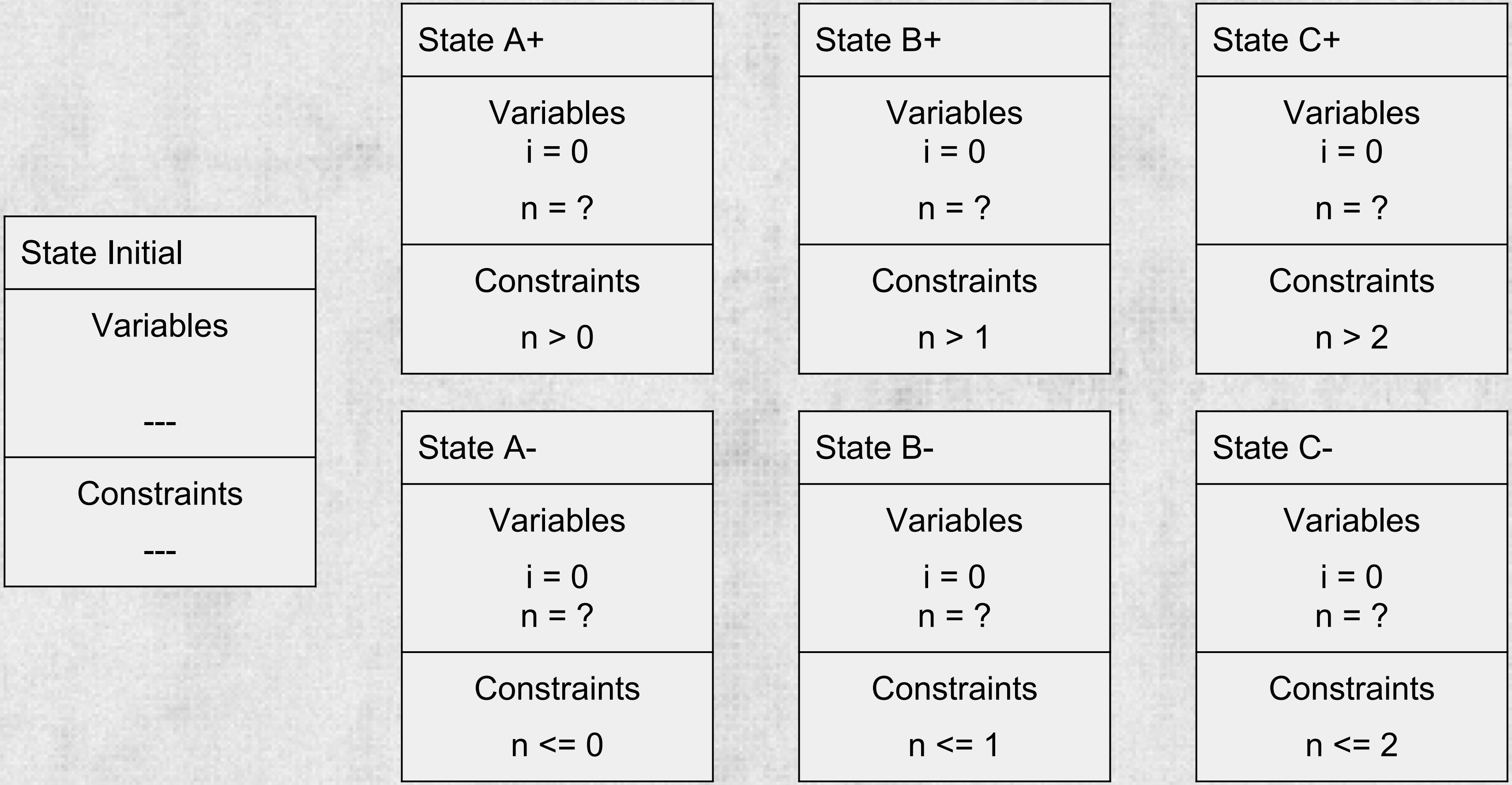
```
void *memcpy(void *dst, void *src, int n)
{
  for (int i = 0; i < n; i++)
     dst[i] = src[i];

  return dst;
}
```

What happens with a symbolic "n"?

Complications...

```
for (int i = 0; i < n; i++) {...}
```



State Initial

Variables

Constraints

State A+

Variables

i = 0

n = ?

Constraints

n > 0

State B+

Variables

i = 0

n = ?

Constraints

n > 1

State C+

Variables

i = 0

n = ?

Constraints

n > 2

State A-

Variables

i = 0

n = ?

Constraints

n <= 0

State B-

Variables

i = 0

n = ?

Constraints

n <= 1

State C-

Variables

i = 0

n = ?

Constraints

n <= 2

Symbolic Summaries

Solution: replace it with a manually written "symbolic summary".

Pro: intelligently reason about conditions

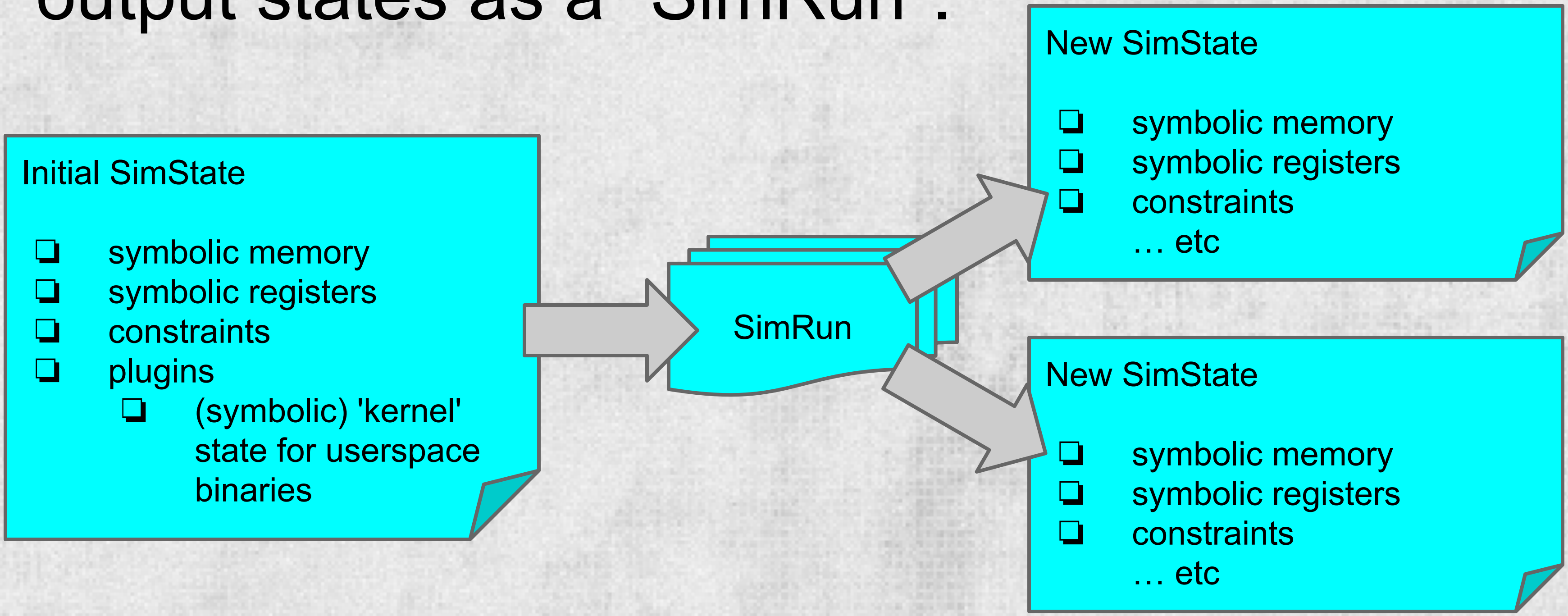
Pro: increased analysis speed

Con: manual implementation

Also used to abstract away system calls.

Useful Abstractions

To support symbolic summaries, we abstract anything that takes an input state and produces output states as a "SimRun".



Initial SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints
- ❑ plugins
  - ❑ (symbolic) 'kernel' state for userspace binaries

SimRun

New SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints

… etc

New SimState

- ❑ symbolic memory
- ❑ symbolic registers
- ❑ constraints

… etc

SimRunForYourLives!

A SimRun can be one of several things:

- ❑ A SimIRSB, to support direct binary analysis
- ❑ A **path** of SimIRSBs, to aid in program slicing
- ❑ A summary of state modifications.

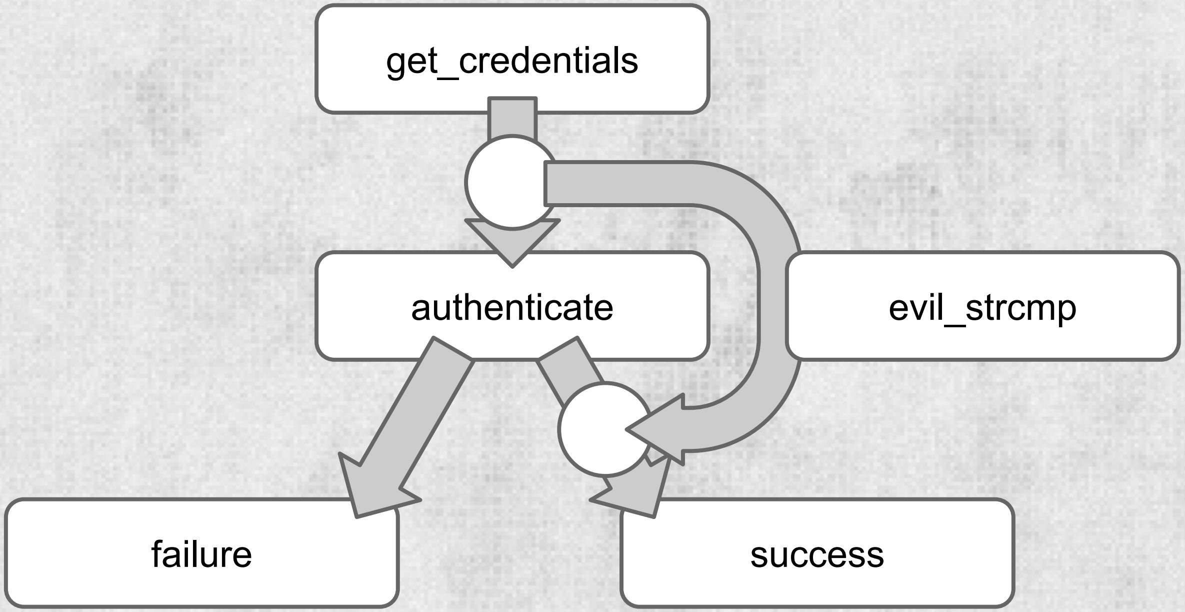
Why?

The SimRun abstraction provides several powerful capabilities:

- Simplifies the analysis
  - most analyses just use SimRun
  - transparenty enable/disable symbolic summaries
- SimRuns can execute in symbolic or concrete mode
  - enables concolic execution on a SimRun-granularity

What do we use this for?

We can leverage all this complex stuff to search for bugs or vulnerabilities! For example, authentication bypass vulnerabilities.



Demo time!

Wow!

We've been gradually releasing stuff!

- So far, the non-symbolic underpinnings.
  - PyVEX (http://github.com/zardus/pyvex)
  - IDALink (http://github.com/zardus/idalink)
  - Other minor, uninteresting things
- More to come!

Questions?
Comments?
Collaboration Ideas?