

A tool for the

# Symbolic Execution

of Linux binaries

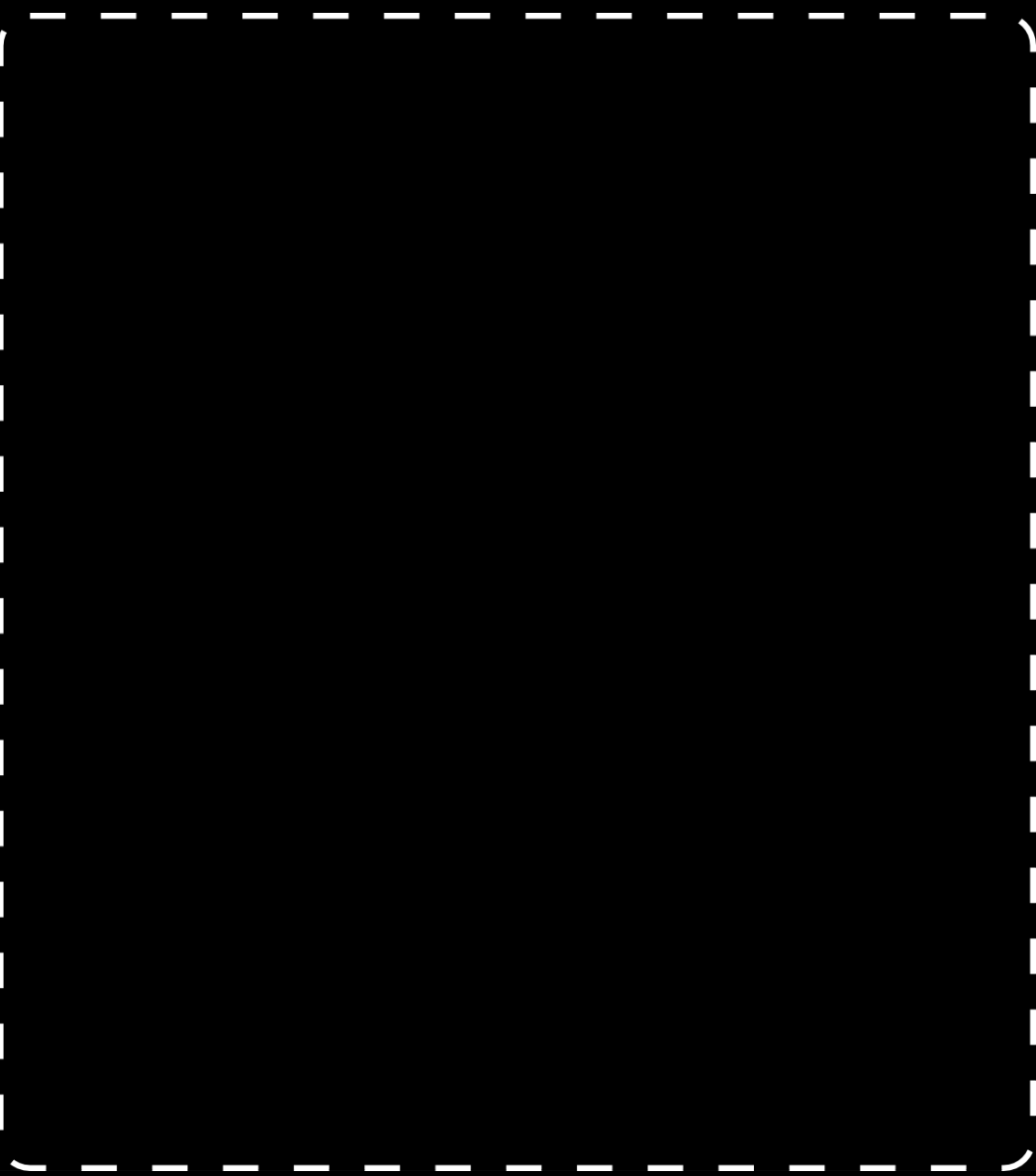
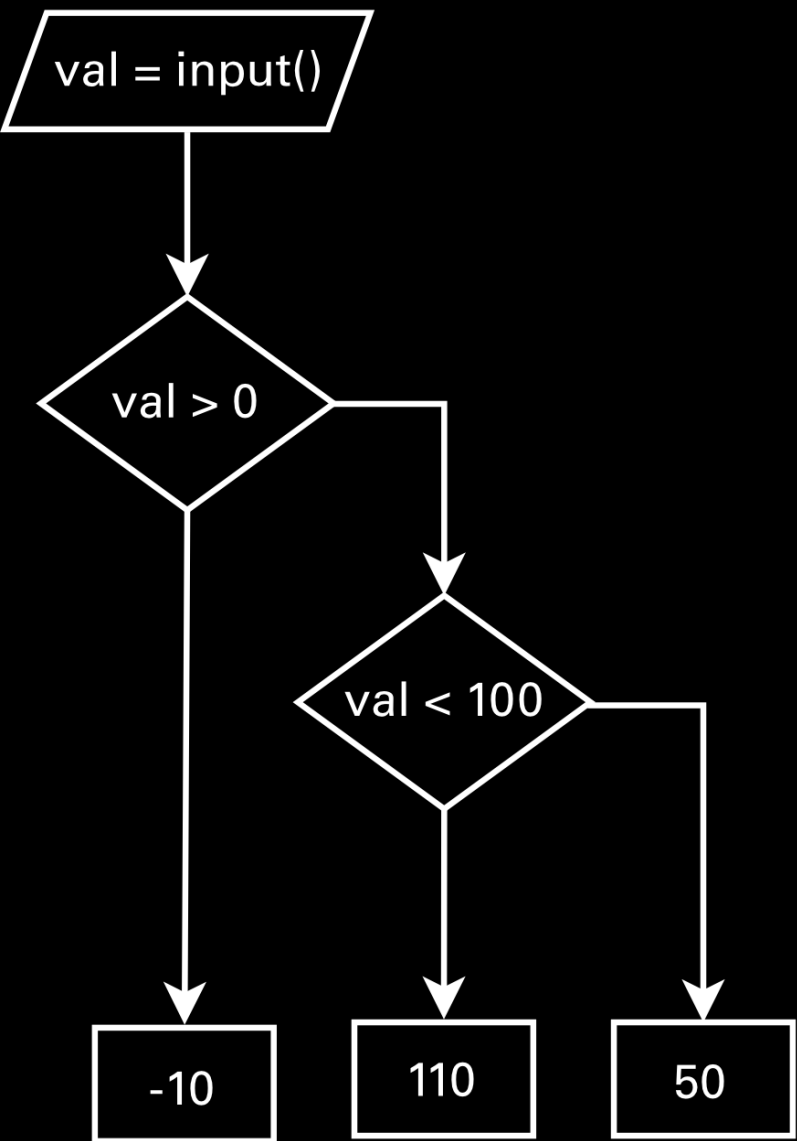


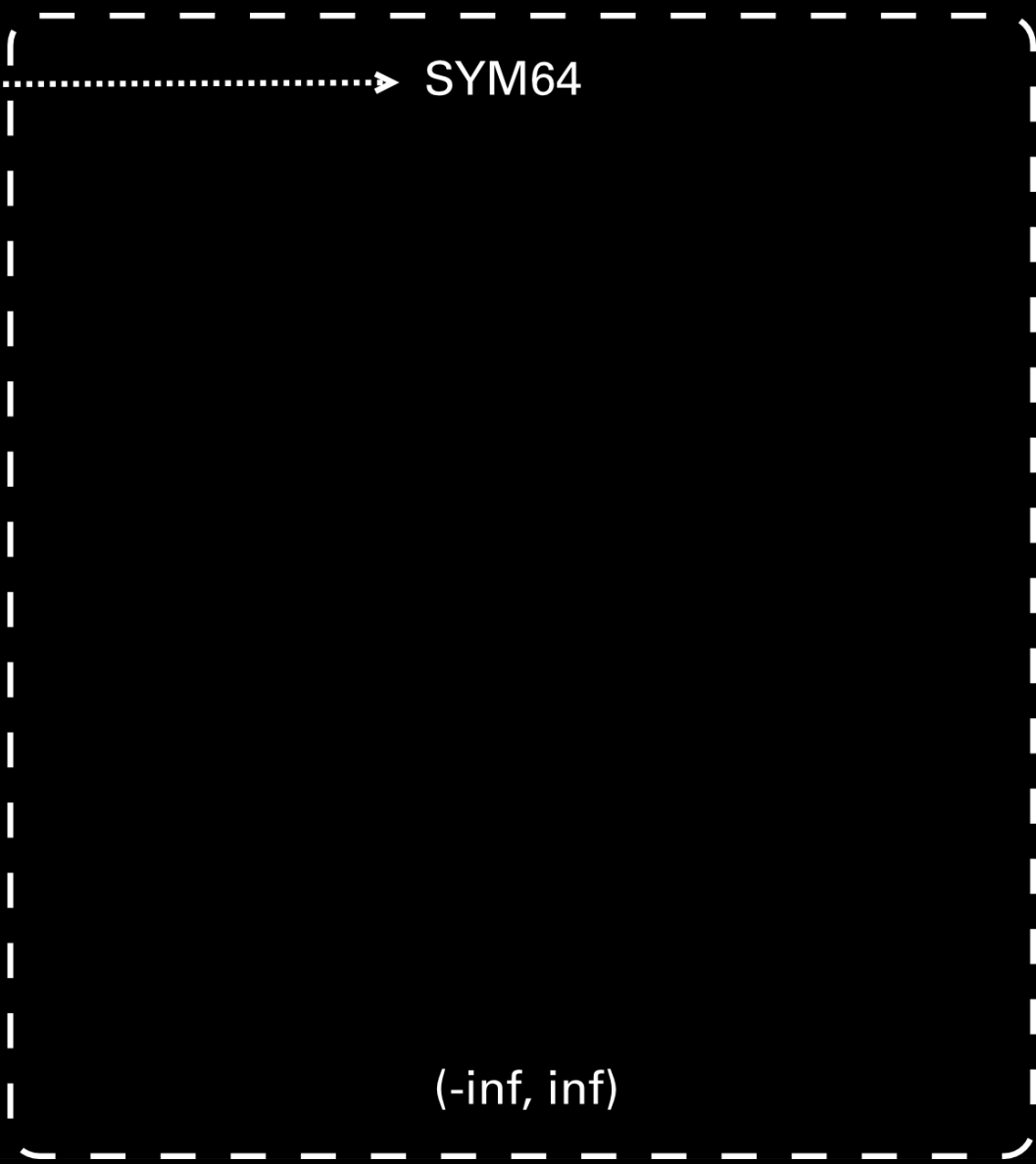
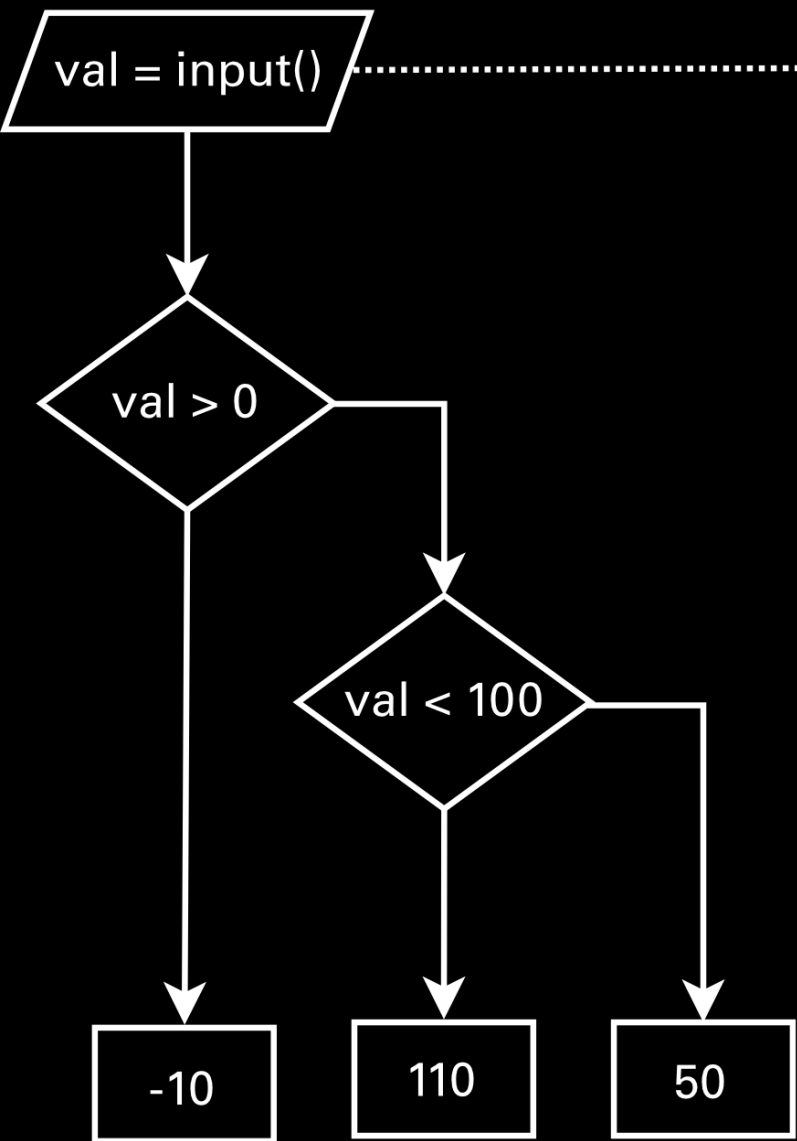
# About Symbolic Execution

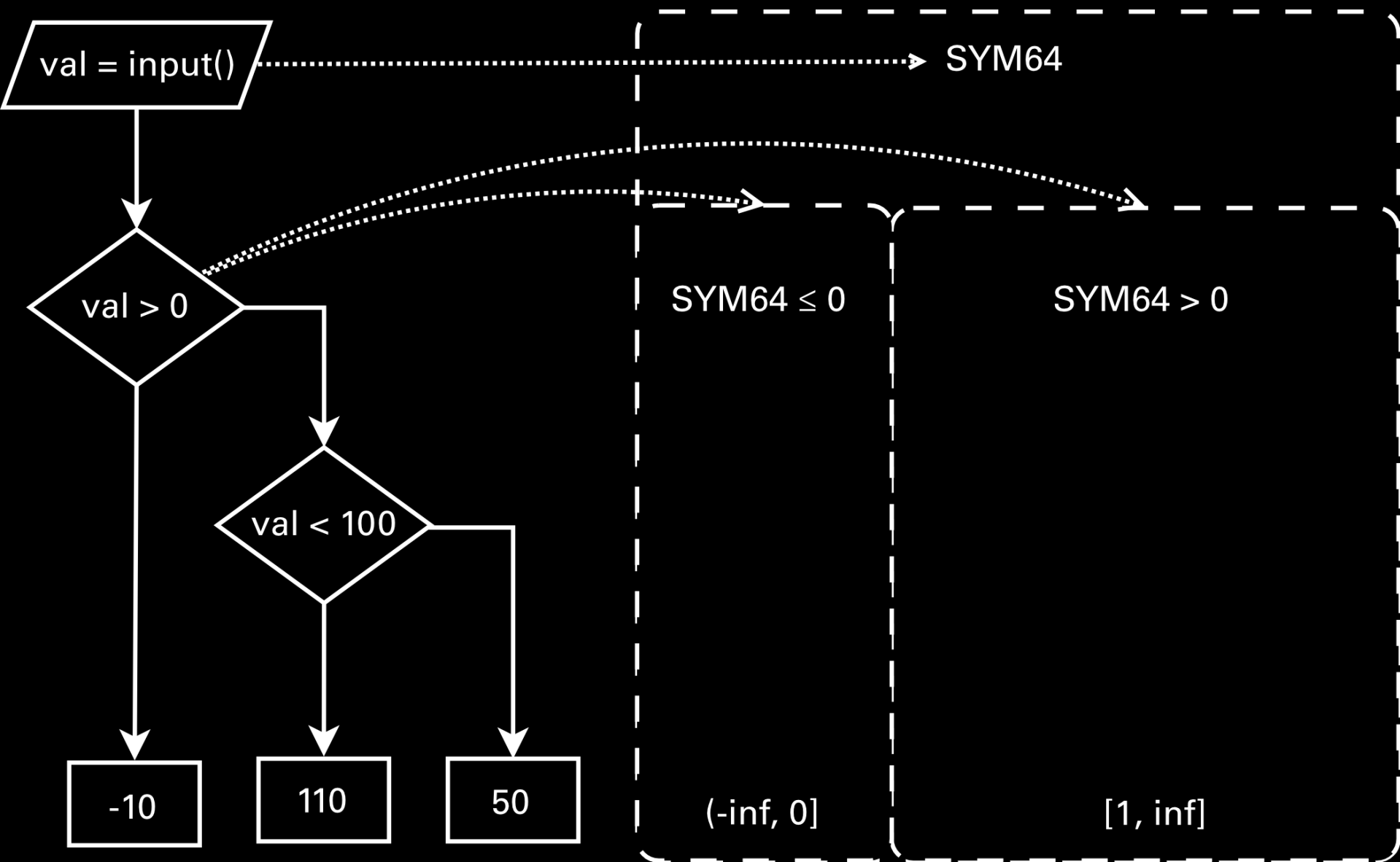
- Dynamically explore all program branches.
- Inputs are considered symbolic variables.
- Symbols remain uninstantiated and become constrained at execution time.
- At a conditional branch operating on symbolic terms, the execution is forked.
- Each feasible branch is taken, and the appropriate constraints logged.

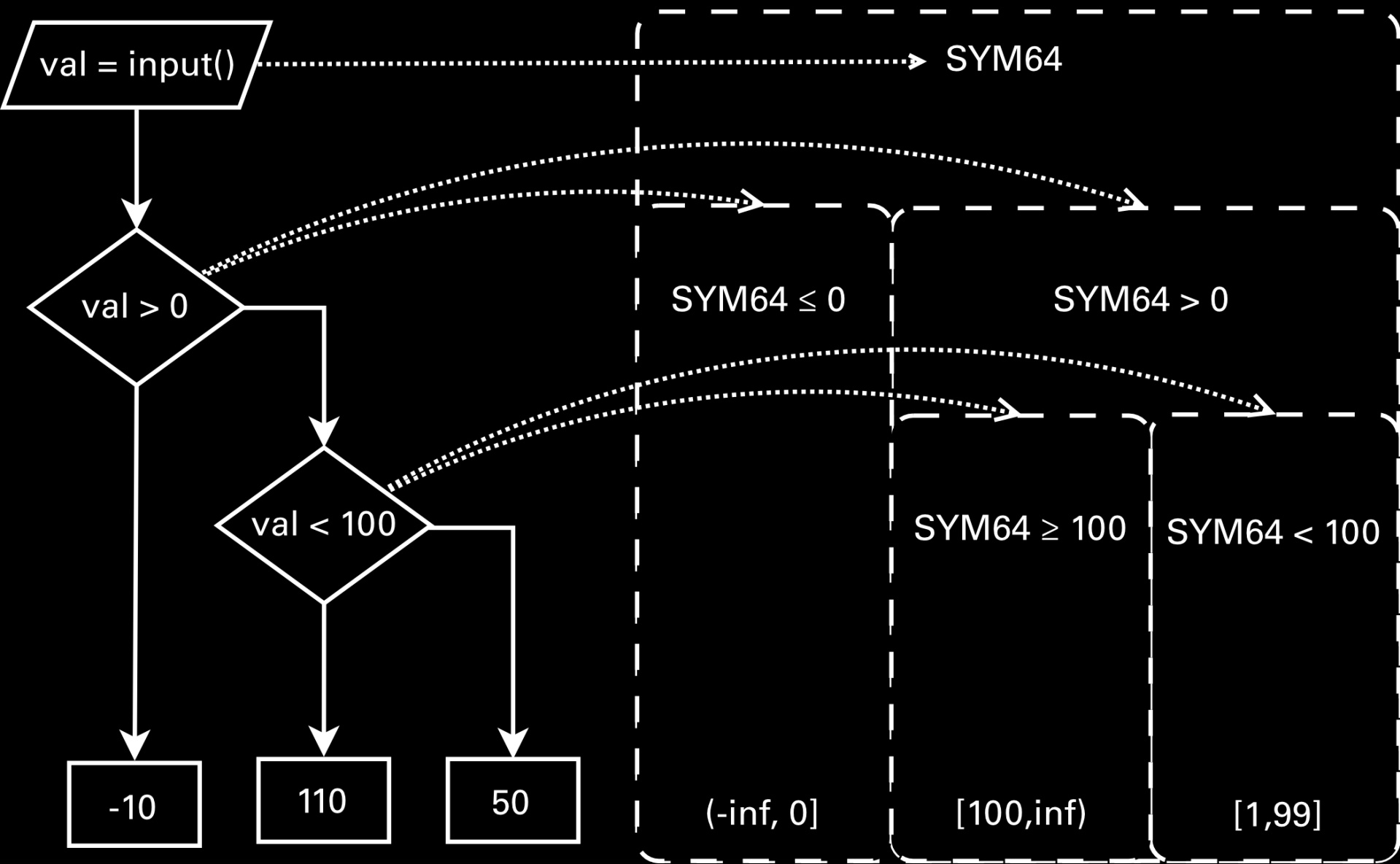
# Input space >> Number of paths

```
int main( ) {  
    int val;  
    read(STDIN, &val, sizeof(val) );  
    if ( val > 0 )  
        if ( val < 100 )  
            do_something( );  
        else  
            do_something_else( );  
}
```









# This is used for:

- Test generation and bug hunting.
- Reason about reachability.
- Worst-Case Execution Time Analysis.
- Comparing different versions of a func.
- Deobfuscation, malware analysis.
- AEG: Automatic Exploit Generation. Whaat?!



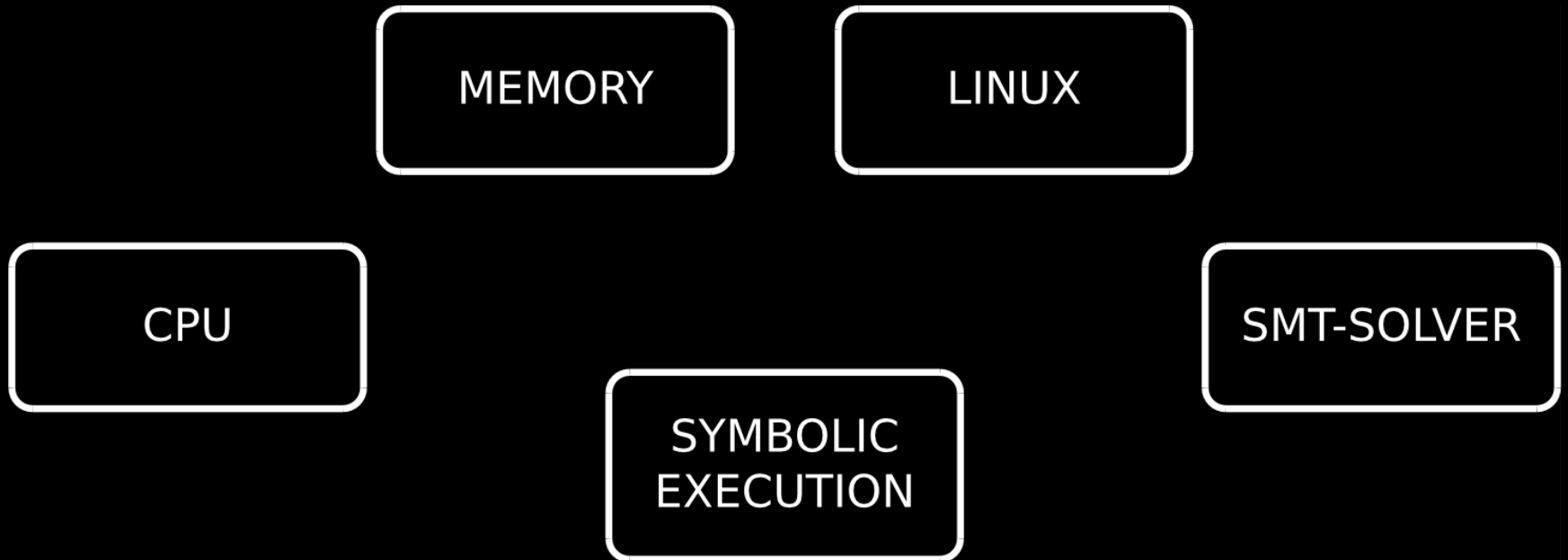
# State of the art

- Lots of academic papers:
  - [2008-12-OSDI-KLEE](#)
  - [Unleashing MAYHEM on Binary](#)
- Several implementations:
  - SymDroid, Cloud9, Pex, jCUTE, Java PathFinder, KLEE, s2e, fuzzball, mayhem, cbass
- Only a few work on binary :
  - libVEX / IL based
  - qemu based

# Our aim

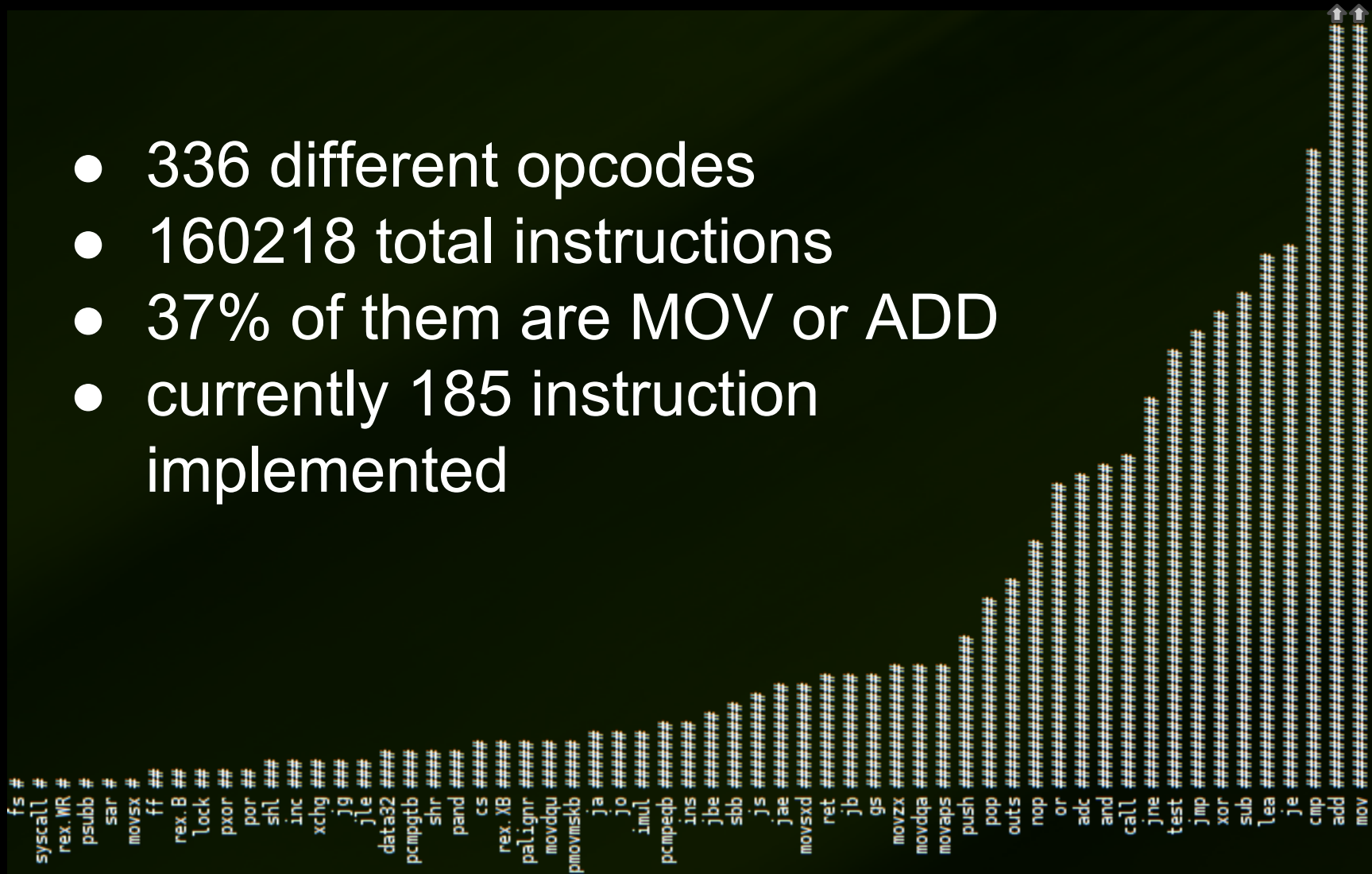
- Emulate x86-64 machine code symbolically.
- Load ELF executables.
- Synthesize any process state as starting point.
- The final code should be readable and easy to extend.
- Use as few dependencies as possible:
  - *pyelftools*, *distorm3* and *z3*
- Analysis state can be saved and restored.
- Workload can be distributed ([dispy](#))

# Basic architecture



# Instructions Frequency in GNU LIBC

- 336 different opcodes
- 160218 total instructions
- 37% of them are MOV or ADD
- currently 185 instruction implemented



# CPU

- Based on distorm3 DecomposeInterface.
- Most instructions are very simple, ex.

```
@instruction
```

```
def DEC(cpu, dest):
```

```
    res = dest.write( dest.read() - 1 )
```

```
    #Affected Flags o..szapc
```

```
    cpu.calculateFlags('DEC', dest.size, res)
```

# Memory



```
class Memory:  
    def mprotect(self, start, size, perms): ...  
    def munmap(self, start, size): ...  
    def mmap(self, addr, size, perms): ...  
    def putchar(self, addr, data): ...  
    def getchar(self, addr): ...
```

# Operating System Model (Linux)

```
class Linux:
    def exe(self, filename, argv=[], envp=[]):...
    def syscall(self, cpu):...

    def sys_open(self, cpu, buf, flags, mode):...
    def sys_read(self, cpu, fd, buf, count):...
    def sys_write(self, cpu, fd, buf, size):...
    def sys_close(self, cpu, fd):...
    def sys_brk(self, cpu, brk):...
```

# Symbols and SMT solver

```
class Solver:
    def getallvalues(self, x, maxcnt = 30):
    def minmax(self, x, iters=10000):
    def check(self):
    def add(self, constraint):
#Symbols factory
    def mkArray(self, size, name ):...
    def mkBool(self, name ):...
    def mkBitVec(self, size, name ):...
```



# Operation over symbols is almost transparent

```
>>> from smtlibv2 import *
>>> s = Solver()
>>> a = s.mkBitVec(32)
>>> b = s.mkBitVec(32)
>>> s.add(a + 2*b > 100)
>>> s.check()
'sat'
>>> s.getvalue(a), s.getvalue(b)
(101, 0)
```

# The glue: Basic Initialization

1. Make Solver, Memory, Cpu and Linux objects.
2. Load ELF binary program in Memory, Initialize cpu registers, initialize stack.

```
solver = Solver()  
mem = SMemory(solver, bits, 12 )  
cpu = Cpu(mem, arch )  
linux = SLinux(solver, [cpu], mem, ... )  
linux.exe("./my_test", argv=[], env=[])
```

# The glue: Basic analysis loop

```
states = ['init.pkl']
while len(states) > 0 :
    linux = load(state.pop())
    while linux.running:
        linux.execute()
        if isinstance( linux.cpu.PC, Symbol):
            vals = solver.getallvalues(linux.
cpu.PC)
            -- generate states for each value --
            break
```

# Micro demo

```
python system.py -h
```

```
usage: system.py [-h] [-sym SYM] [-stdin STDIN]  
[-stdout STDOUT]
```

```
[-stderr STDERR] [-env ENV]
```

```
PROGRAM ...
```

```
python system.py -sym stdin my_prog
```

```
stdin:
```

```
PDF-1.2+++++
```

# Symbolic inputs.

We need to mark which part of the environment is symbolic:

- `STDIN`: a file partially symbolic. Symbols marked with “+”
- `STDOUT` and `STDERR` are placeholders.
- `ARGV` and `ENV` can be symbolic

# A toy example

```
int main(int argc, char* argv[], char* envp[]) {
    char buffer[0x100] = {0};
    read(0, buffer, 0x100);
    if (strcmp(buffer, "ZARAZA") == 0 )
        printf("Message: ZARAZA!\n");
    else
        printf("Message: Not Found!\n");
    return 0;
}
```

# Conclusions, future work

- Push all known optimizations: solver cache, implied values, Redundant State Elimination, constraint independence, KLEE-like cex cache, symbol simplification.
- Add more cpu instructions (fpu, simd).
- Improve Linux model, add network.
- Implement OSX loader and os model.
- <https://github.com/feliam/pysyemu>

Gracias.

Contacto:

[feliam@binamuse.com](mailto:feliam@binamuse.com)