# Detecting Metamorphic Computer Viruses using Supercompilation

Alexei Lisitsa and Matt Webster
Department of Computer Science
University of Liverpool, UK

TCV 2008
Nancy, France, May 2008

# Structure of the Presentation

- Introduction
  - Metamorphic computer viruses
  - Supercompilation
- Interpreter of Intel 64
  - Proving equivalence of programs
  - Proving non-equivalence of programs
- Detection of metamorphic computer viruses
- Conclusion

# Metamorphic Computer Viruses

- Metamorphic computer viruses
  - Change their syntax
  - Keep their behaviour (semantics) constant
  - Are able to evade detection by signature scanning
- Examples: Zmorph, Bistro, Apparition, ...
- Undetectable metamorphic computer viruses exist!
  - Chess & White (2000) - existence proof
  - Filiol & Josse (2007) - constructive proof

# Supercompilation

- Supercompilation = *Super*vised *compilation*
  - Developed by Valentin Turchin (1970s)
  - An approach to program transformation
    - Improve efficiency of functional programs
    - Has been used for verification (Lisitsa & Nemytykh, 2007)
- SCP4 (Nemytykh, Turchin)
  - The most advanced supercompiler
  - Works with the recursive functions algorithmic language (Refal)
  - Other supercompilers exist
    - Java, Haskell

# Supercompilation (2)

- How does supercompilation work?
  - A program and its parameter are taken as input
  - A graph of all possible states is constructed
    - This may be an infinite graph
    - This stage is called *unfolding*
  - This tree is analysed
    - Using *generalisation*, this tree is *folded* into another tree
    - This second tree represents the configurations of the parameterised program
  - Infinite tree of states → Finite tree of states
- Therefore, supercompilation can be used...
  - ... for program specialisation and optimisation

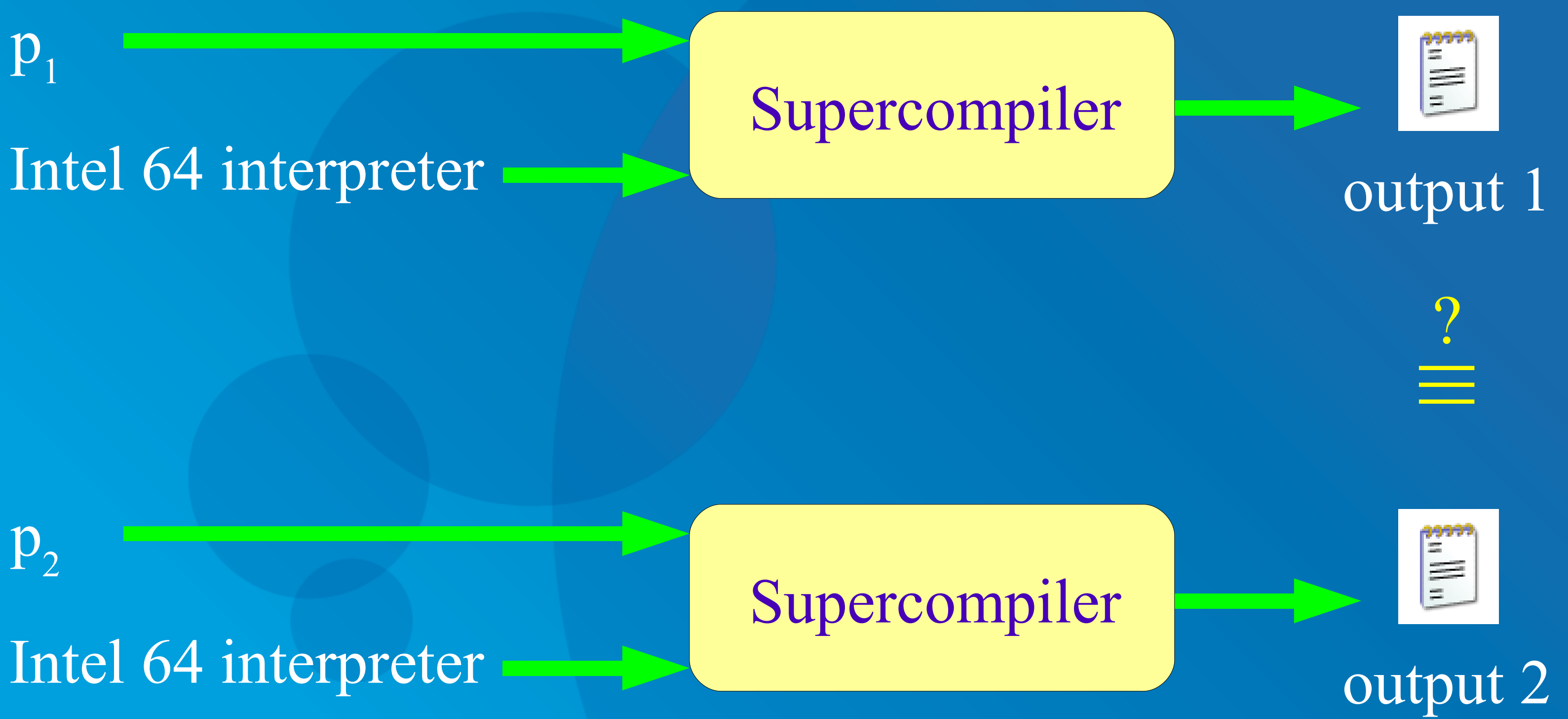
# Intel 64 Interpreter

- Programmed in Refal

| Instruction type | Refal clause |
| --- | --- |
| | `mov {` |
| mov eax, *n* | `(eax (const e.1))(eax e.2)(ebx e.3)(ecx e.4)(Zflag e.5) = (eax e.1)(ebx e.3)(ecx e.4)(Zflag e.5);` |
| mov eax, ebx | `(eax (reg ebx))(eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4) = (eax e.2)(ebx e.2)(ecx e.3)(Zflag e.4);` |
| ... | `...` |
| | `}` |

- Other instructions implemented so far
  - jumps (JMP), conditionals (CMP), conditional jumps (JE)

# Proving Program Equivalence



# Proving Program Equivalence (2)

$p_1$

```
mov eax, 0
mov ebx, 1
cmp eax, ebx
```

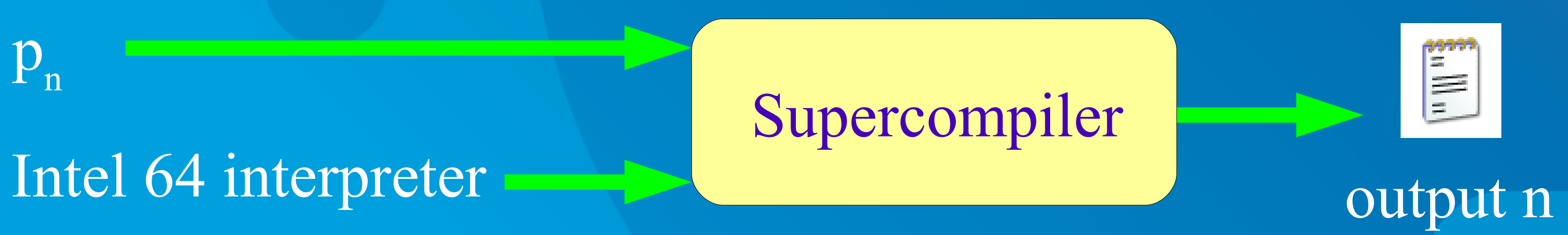
$p_2$

```
jmp 1
label 1:
mov ebx, 1
mov eax, ebx
mov eax, ecx
mov eax, 0
jmp 2
mov eax, ecx
jmp 1
label 2:
cmp eax, ebx
```

$p_3$

```
mov eax, 1
mov ebx, 1
cmp eax, ebx
je 1
mov eax, 5
label 1:
mov eax, 0
cmp eax, ebx
je 1
mov eax, 0
```

- Supercompile each program
- Check the result of supercompilation
- If they are the same
  - ... then the programs are equivalent



# Proving Program Equivalence (3)

$p_1$

```
mov eax, 0
mov ebx, 1
cmp eax, ebx
```

$p_2$

```
jmp 1
label 1:
mov ebx, 1
mov eax, ebx
mov eax, ecx
mov eax, 0
jmp 2
mov eax, ecx
jmp 1
label 2:
cmp eax, ebx
```

$p_3$

```
mov eax, 1
mov ebx, 1
cmp eax, ebx
je 1
mov eax, 5
label 1:
mov eax, 0
cmp eax, ebx
je 1
mov eax, 0
```

$\equiv$

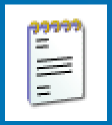
- Result of supercompilation

```
$ENTRY Go {
 (e.101) (e.102) (e.103) (e.104)  =
 (eax 0) (ebx 1) (ecx e.103) (Zflag 0) ;
}
```

$p_n$

Intel 64 interpreter

Supercompiler



output n

# Proving Program Non-Equivalence

$p_1$

```
mov eax, 0
mov ebx, 1
cmp eax, ebx
```

$p_2$

```
mov eax, 1
mov ebx, 1
cmp eax, ebx
je 1
mov eax, 5
label 1:
mov eax, 0
cmp eax, ebx
je 1
mov eax, 1
```

- Supercompile each program
- Check the result of supercompilation
- If they are not the same
  - … then the programs may not be equivalent

# Proving Program Non-Equivalence (2)

$p_1$

```
mov eax, 0
mov ebx, 1
cmp eax, ebx
```

$\neq$

$p_4$

```
mov eax, 1
mov ebx, 1
cmp eax, ebx
je 1
mov eax, 5
label 1:
mov eax, 0
cmp eax, ebx
je 1
mov eax, 1
```

- Result of supercompilation

$p_1$

```
$ENTRY Go {
 (e.101) (e.102) (e.103) (e.104)  =
 (eax 0) (ebx 1) (ecx e.103) (Zflag 0) ;
}
```

$p_4$

```
$ENTRY Go {
 (e.101) (e.102) (e.103) (e.104)  =
 (eax 1) (ebx 1) (ecx e.103) (Zflag 0) ;
}
```

$p_n$ → Supercompiler

Intel 64 interpreter → Supercompiler

Supercompiler → output n

# Supercompilation for Detection

- Metamorphic computer virus variants must have equivalent behaviour
  - We can prove program equivalence using supercompilation
  - Therefore, we can use supercompilation for detection
- We assume that the suspect code and signature are already prepared
  - Then, we can use supercompilation to prove program equivalence

# Supercompilation for Detection

- Limitations
  - The supercompilation algorithm cannot normalise all equivalent programs to the same syntactic form
    - Undecidable problem!
  - False negatives are possible
    - Some code is not analysable by supercompiler
- Good news
  - False positives are unlikely, or even impossible
    - This needs to be investigated formally
    - Perhaps this is not so hard:
      - Supercompilation is built upon formal foundations

# Conclusion

- Supercompilation can be used to detect metamorphic computer viruses
- Future work:
  - Extend our interpreter for Intel 64
    - Try out our technique on realistic metamorphic virus code
  - Discover the bounds of detection by supercompilation
    - Which cases, in general, allow detection?
    - Which cases don't?
    - Is detection-by-supercompilation formally correct?

# End of Presentation

- Any questions?