Program Synthesis in Reverse Engineering

Rolf Rolles
Möbius Strip Reverse Engineering

http://www.msreverseengineering.com

No Such Conference Keynote Speech
November 19th, 2014

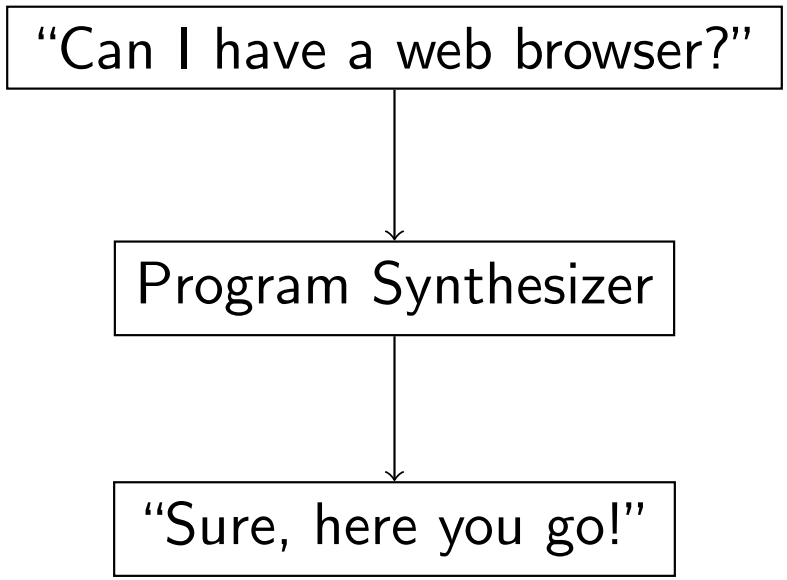
Program Synthesis in Reverse Engineering

Overview

- **Program synthesis** is an academic discipline devoted to creating programs automatically, given an expectation of how the program should behave.
- We apply and adapt existing academic work to automate tasks in reverse engineering.

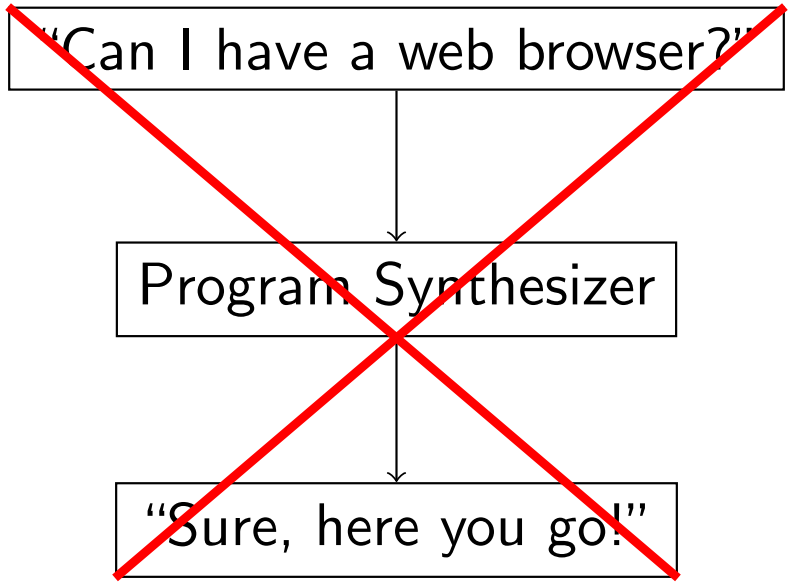
Program Synthesis

What it is Not



Program Synthesis

What it is Not



Program synthesis:

- Requires precise behavioral specifications
- Operates on small scales
- Is primarily researched on loop-free programs

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x >= 0} \\ \texttt{-x} & \text{otherwise} \end{cases}$$

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} >= 0 \\ \texttt{-x} & \text{otherwise} \end{cases}$$

- Create a harness for testing `abs(x)` exhaustively:

```
int main(int, char **) {
  for(int x = 0; x != MAX_INT; ++x) {
    int r = abs(x);
    if((x >= 0 && r != x) || r != -x)
      return -1;
  }
  return 0;
}
```

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} >= 0 \\ \texttt{-x} & \text{otherwise} \end{cases}$$

- Create a harness for testing `abs(x)` exhaustively:

```
int main(int, char **) {
  for(int x = 0; x != MAX_INT; ++x) {
    int r = abs(x);
    if((x >= 0 && r != x) || r != -x)
      return -1;
  }
  return 0;
}
```

- Enumerate every possible function `abs(int x)`:

```
int abs(int x) { return -x; }
```

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} >= 0 \\ \texttt{-x} & \text{otherwise} \end{cases}$$

- Create a harness for testing `abs(x)` exhaustively:

```
int main(int, char **) {
  for(int x = 0; x != MAX_INT; ++x) {
    int r = abs(x);
    if((x >= 0 && r != x) || r != -x)
      return -1;
  }
  return 0;
}
```

- Enumerate every possible function `abs(int x)`:

```
int abs(int x) { return ~x; }
```

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} >= 0 \\ \texttt{-x} & \text{otherwise} \end{cases}$$

- Create a harness for testing `abs(x)` exhaustively:

```
int main(int, char **) {
  for(int x = 0; x != MAX_INT; ++x) {
    int r = abs(x);
    if((x >= 0 && r != x) || r != -x)
      return -1;
  }
  return 0;
}
```

- Enumerate every possible function `abs(int x)`:

```
int abs(int x) { return -(x+0); }
```

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} >= 0 \\ \texttt{-x} & \text{otherwise} \end{cases}$$

- Create a harness for testing `abs(x)` exhaustively:

```
int main(int, char **) {
  for(int x = 0; x != MAX_INT; ++x) {
    int r = abs(x);
    if((x >= 0 && r != x) || r != -x)
      return -1;
  }
  return 0;
}
```

- Enumerate every possible function `abs(int x)`:

```
int abs(int x) { return ~(x+0); }
```

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} >= 0 \\ \texttt{-x} & \text{otherwise} \end{cases}$$

- Create a harness for testing `abs(x)` exhaustively:

```
int main(int, char **) {
  for(int x = 0; x != MAX_INT; ++x) {
    int r = abs(x);
    if((x >= 0 && r != x) || r != -x)
      return -1;
  }
  return 0;
}
```

- Enumerate every possible function `abs(int x)`:

```
int abs(int x) { return -(x+1); }
```

Program Synthesis

The Simplest Possible Program Synthesizer

- Starting from a behavioral specification:

$$\texttt{int abs(int x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} >= 0 \\ \texttt{-x} & \text{otherwise} \end{cases}$$

- Create a harness for testing `abs(x)` exhaustively:

```
int main(int, char **) {
  for(int x = 0; x != MAX_INT; ++x) {
    int r = abs(x);
    if((x >= 0 && r != x) || r != -x)
      return -1;
  }
  return 0;
}
```

- Enumerate every possible function `abs(int x)`:

```
int abs(int x) { return ~(x+1); }
```

Program Synthesis

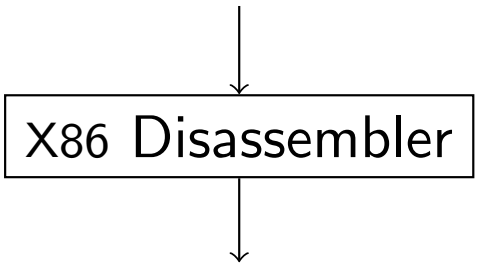
The Simplest Possible Program Synthesizer

- The brute-force synthesizer:
  - Is extremely slow, and
  - Explores an infinite number of programs.
- We could improve the situation by carefully choosing which programs to generate.
- Modern program synthesis goes further and does better.

Ingredients

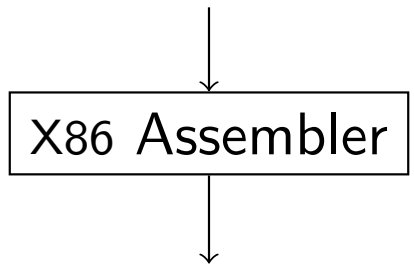
X86 Disassembly and Assembly

```
28 D8
0F B6 DB
81 C3 BB BB BB BB
01 D8
```




```
sub bl, bl
movzx ebx, bl
add ebx, 0BBBBBBBBh
add eax, ebx
```

```
sub bl, bl
movzx ebx, bl
add ebx, 0BBBBBBBBh
add eax, ebx
```

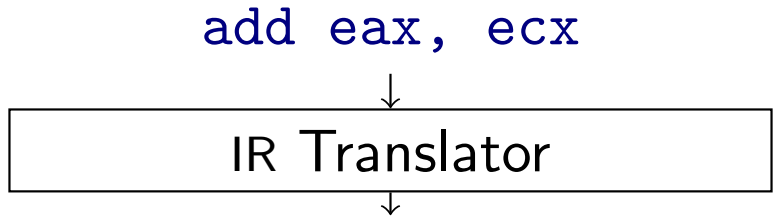



```
28 D8
0F B6 DB
81 C3 BB BB BB BB
01 D8
```

Given bytes, a **disassembler** produces X86 assembly. An **assembler** does the opposite.

Ingredients

IR Translation

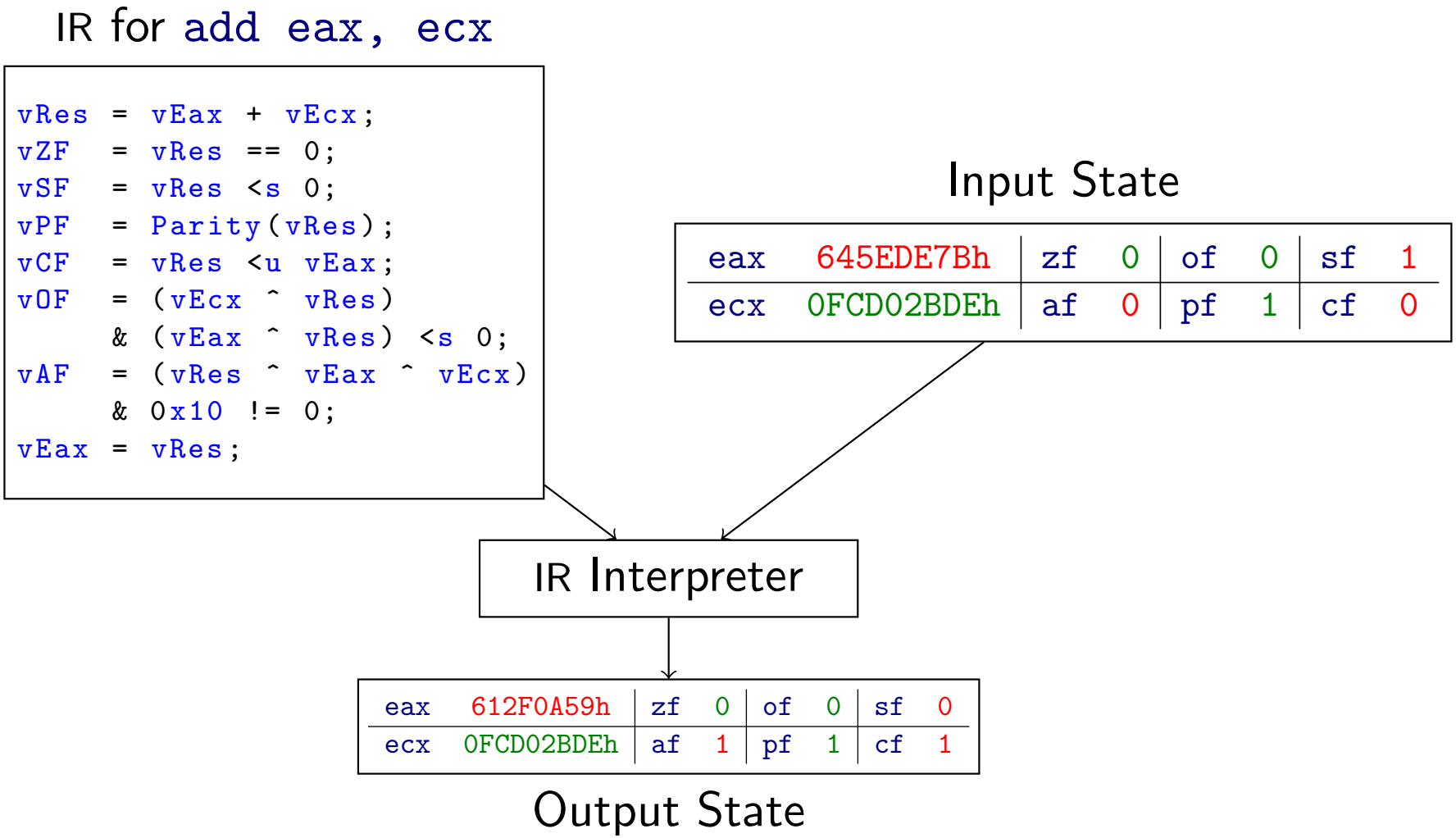



```
vRes = vEax + vEcx;
vZF  = vRes == 0;
vSF  = vRes <s 0;
vPF  = Parity(vRes);
vCF  = vRes <u vEax;
vOF  = (vEcx ^ vRes)
     & (vEax ^ vRes) <s 0;
vAF  = (vRes ^ vEax ^ vEcx)
     & 0x10 != 0;
vEax = vRes;
```

IR translators convert instructions to a symbolic representation of their actual effects when executed.

Ingredients

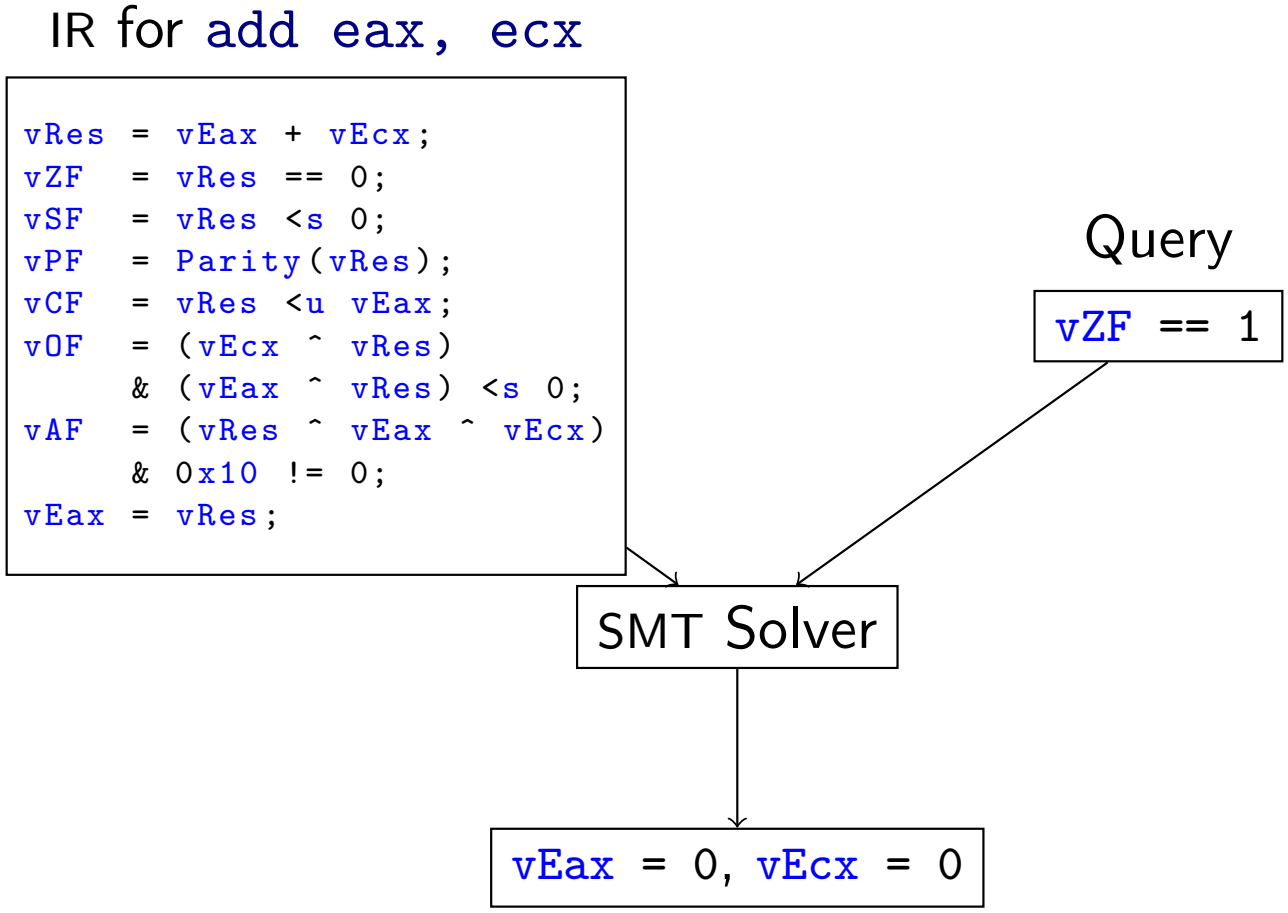
IR Interpreter



The **IR interpreter** executes IR statements in an input **state**, producing an output **state**.

Ingredients

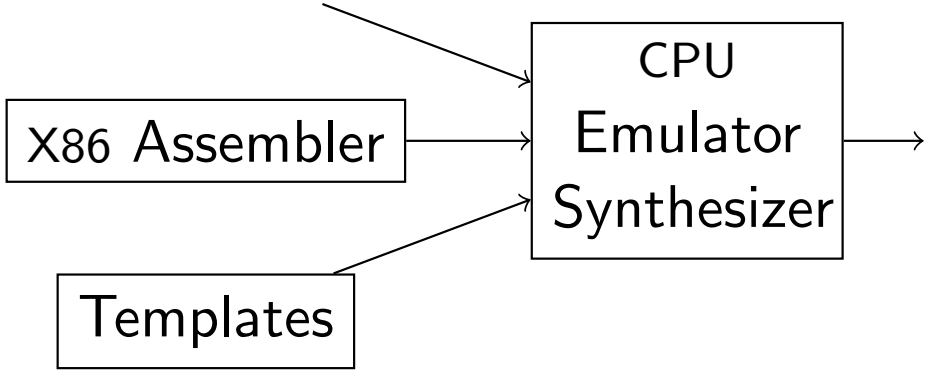
SMT Integration



- **SMT solvers** can answer questions about the values of variables and memory within IR sequences.
- We queried for a **model** of `eax` and `ecx` where `vZF == 1`.

Program Synthesis in Reverse Engineering

CPU Emulator Synthesis



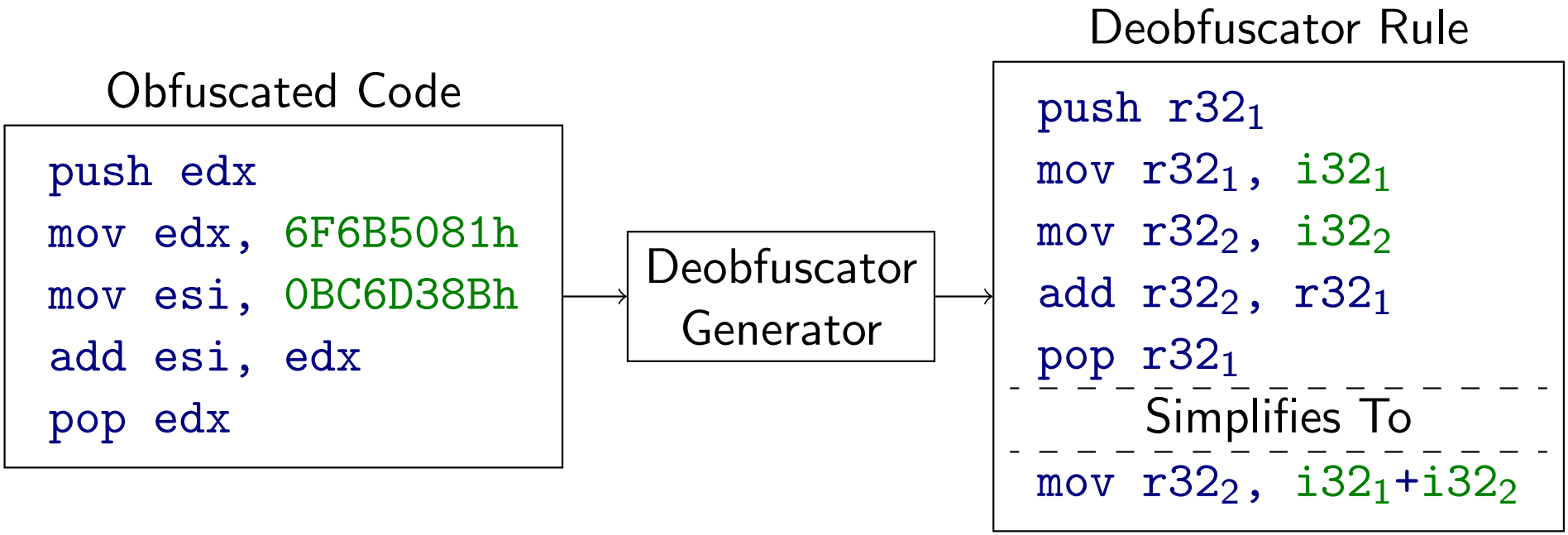
CPU Emulator Logic

```
vRes = vEax + vEcx;
vZF  = vRes == 0;
vSF  = vRes <s 0;
vPF  = Parity(vRes);
vCF  = vRes <u vEax;
vOF  = (vEcx ^ vRes)
     & (vEax ^ vRes) <s 0;
vAF  = (vRes ^ vEax ^ vEcx)
     & 0x10 != 0;
vEax = vEax + vEcx;
```

- We present a re-designed version of [2] for generating CPU emulators.

Program Synthesis in Reverse Engineering

Peephole Superdeobfuscation

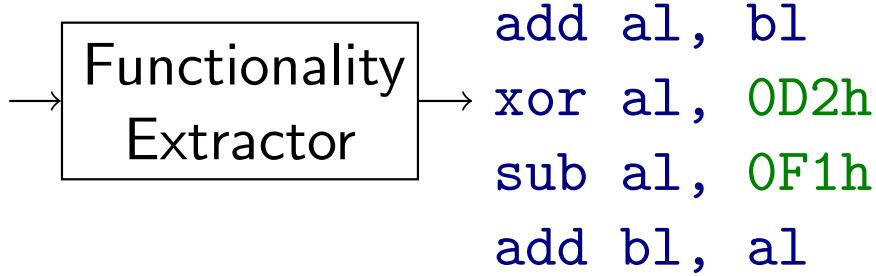


- We apply the ideas of [1] to automatically create deobfuscators for certain obfuscators.

Program Synthesis in Reverse Engineering

Metamorphic Extraction

```
add al, 0B7h
sub al, 90h
push cx
mov cl, bl
add al, bl
pop cx
add al, 90h
sub al, 0B7h
push ebx
mov ebx, 8716AEF1h
push ecx
push eax
xor dword ptr [esp], ebx
; continued
```



- We apply a technique from [3] to automatically recover metamorphically-generated code sequences.

CPU Emulator Synthesis

Overview

- CPU emulators require accurate descriptions of the operation of the X86 processor.
- Sadly, the Intel manuals are at best vague, at worst wrong.

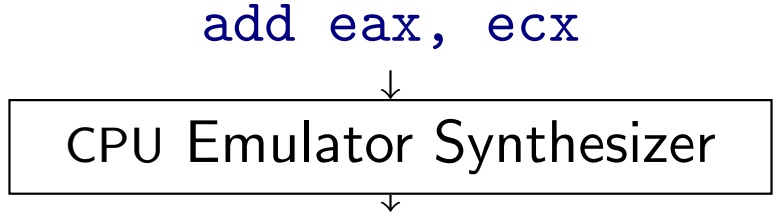
```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
WRONG:          AL ← AL + 6;        AX ← AX + 0x106
WRONG:          AH ← AH + 1;
                AF ← 1;
                CF ← 1;
                AL ← AL AND 0FH;
            ELSE
                AF ← 0;
                CF ← 0;
                AL ← AL AND 0FH;
        FI;
FI;
```

- That's from the first instruction in the manual.

CPU Emulator Synthesis

Overview

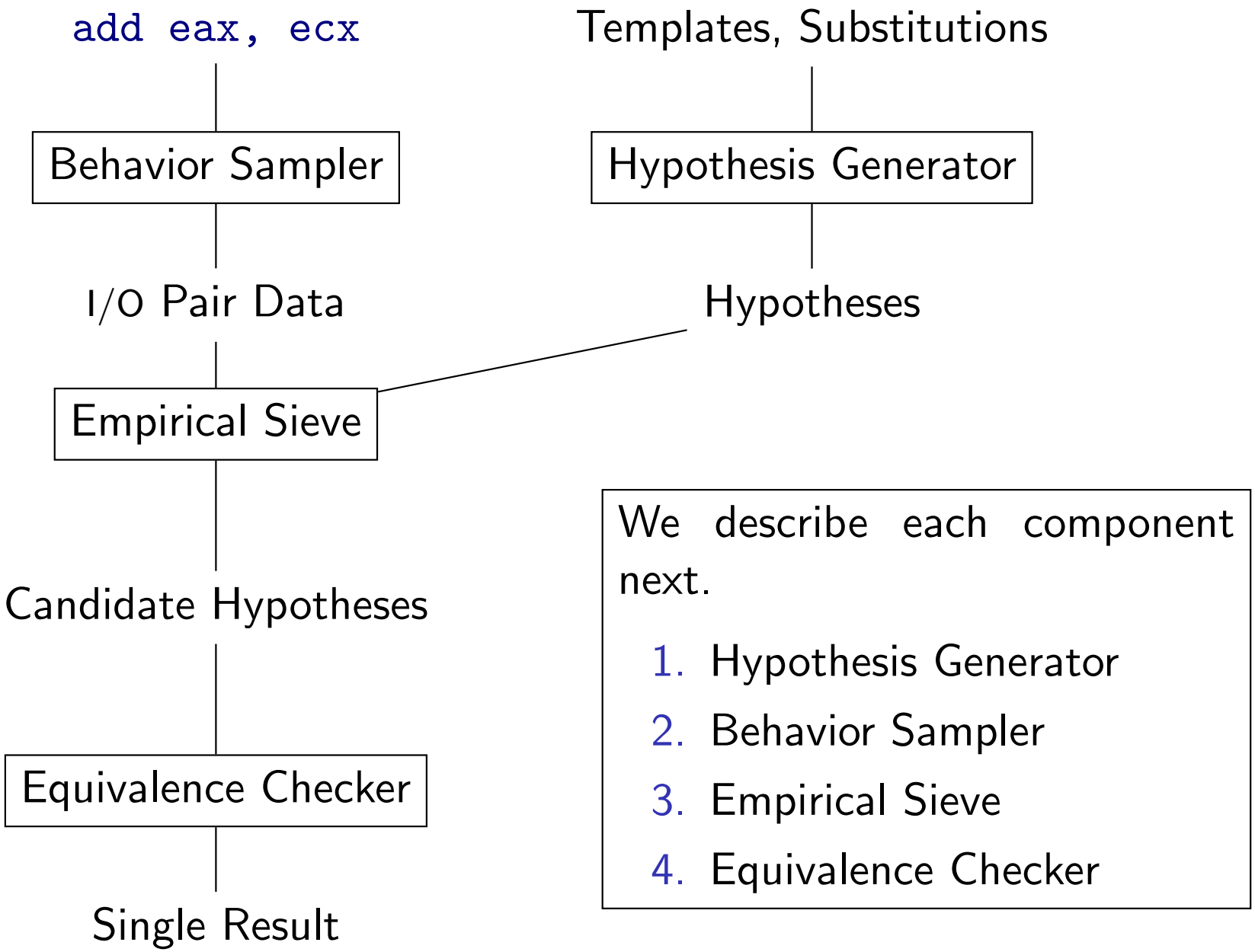
- Idea: use program synthesis to find instruction descriptions.




```
vRes = vEax + vEcx;
vZF  = vRes == 0;
vSF  = vRes <s 0;
vPF  = Parity(vRes);
vCF  = vRes <u vEax;
vOF  = (vEcx ^ vRes)
     & (vEax ^ vRes) <s 0;
vAF  = (vRes ^ vEax ^ vEcx)
     & 0x10 != 0;
vEax = vEax + vEcx;
```

CPU Emulator Synthesis

Overview



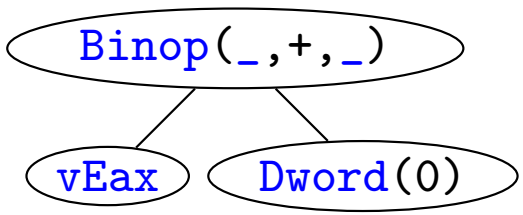
We describe each component next.

1. Hypothesis Generator
2. Behavior Sampler
3. Empirical Sieve
4. Equivalence Checker

CPU Emulator Synthesis

Hypothesis Generator

We generate a family of expressions from a **template expression** and a list of **substitutions**.



Tree representation of `vEax + 0`

Token	Replace With
`+`	`+,-,\&,|,^`
`vEax`	`vEax,vEcx`

Specified Substitutions

```
vEax + 0   vEax - 0   vEax & 0   vEax | 0   vEax ^ 0
vEcx + 0   vEcx - 0   vEcx & 0   vEcx | 0   vEcx ^ 0
```

The $5 * 2 = 10$ expressions generated from the above

CPU Emulator Synthesis

Hypothesis Generator

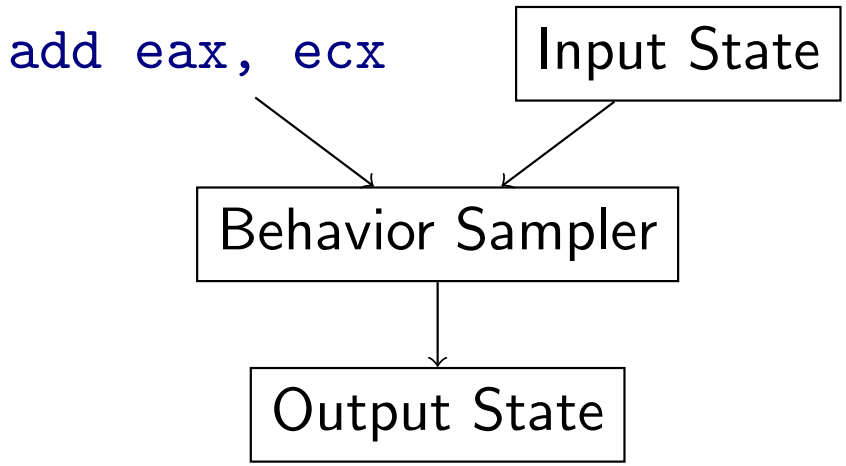
- A **result location** is a location modified by an instruction.
- A **hypothesis** is an IR expression of equality between a result location variable and a generated expression.
- For the instruction `add eax, ebx`, the result locations are $\texttt{vEax}_{out}$, $\texttt{vZF}_{out}$, $\texttt{vSF}_{out}$, $\texttt{vPF}_{out}$, $\texttt{vCF}_{out}$, $\texttt{vOF}_{out}$, and $\texttt{vAF}_{out}$.

$\texttt{vZF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vZF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vSF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vSF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vPF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vPF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vCF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vCF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vOF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vOF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0
$\texttt{vAF}_{out}$ == (vEax + vEbx) <s 0	$\texttt{vAF}_{out}$ == ((vEax + vEbx) ^ vEax) >u 0

Some hypotheses for `add eax, ebx`

CPU Emulator Synthesis

Behavior Sampler



Given a set of register and flag values as input:

1. JIT assemble the instruction
2. Set the processor registers and flags to the input state
3. Execute the instruction
4. Collect the values of the registers and flags as output

These **I/O pairs** are samples of the instruction's behavior.

CPU Emulator Synthesis

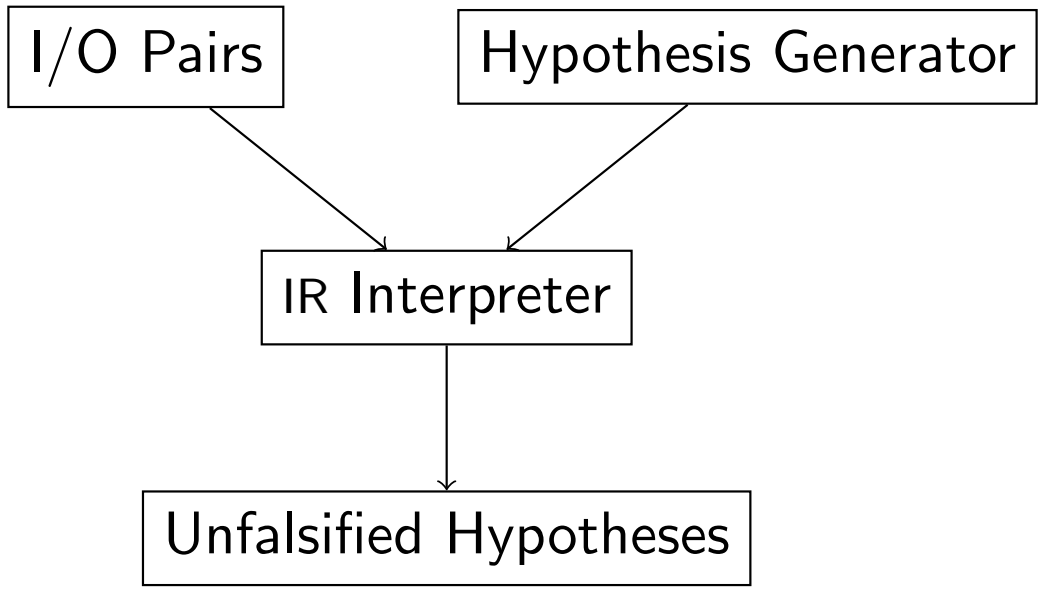
Empirical Sieve

$\texttt{eax}_{in} = 0$		`0 == (0 == 0)`
$\texttt{ecx}_{in} = 0$	$\texttt{CF}_{out}$ `== (vEax == vEcx)`	`0 == 1`
$\texttt{cf}_{out} = 0$	Hypothesis	`false`
I/O Data		Evaluation

- If a hypothesis is `false` in any I/O pair, it cannot describe the instruction's behavior, so we discard it.
- In the figure above, $\texttt{CF}_{out}$ `== (vEax == vEcx)` is an invalid hypothesis because it evaluates to `false` for the pair listed.

CPU Emulator Synthesis

Empirical Sieve



- We generate many hypotheses, test them against the I/O pairs, and discard any that evaluate to `false`.
- Multiple unfalsified hypotheses may remain after the empirical sieve. The next component removes more of them.

CPU Emulator Synthesis

Equivalence Checker

- **Equivalence checking** uses an SMT solver to determine whether two expressions evaluate equally for all inputs. If not, it finds an input that causes a different evaluation.

`(vEax+vEcx) == 0`	`(vEcx+vEax+vEcx+vEax) == 0`
Expression #1	Expression #2

Two hypotheses for $\texttt{ZF}_{out}$.

- We query an SMT solver as to the satisfiability of `((vEax+vEcx) == 0) != ((vEcx+vEax+vEcx+vEax) == 0)`.
  - If UNSAT, the hypotheses always behave the same.
  - IF SAT, the solution contains values for `vEax` and `vEcx` that cause the hypotheses to evaluate differently.

CPU Emulator Synthesis

Equivalence Checker

- The SMT solver reports satisfiability, and gives a model where the first hypothesis for $\texttt{ZF}_{out}$ evaluates to `0`, and the other to `1`.

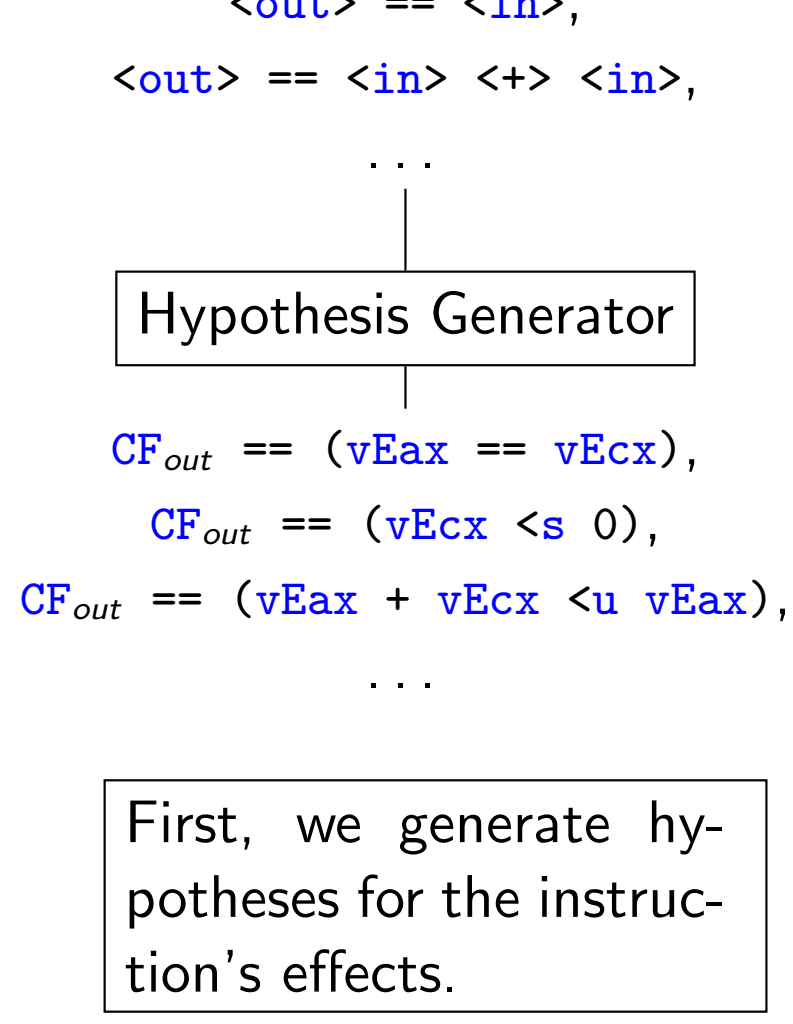
$$\texttt{eax}_{in} = \texttt{3d800000h}$$
$$\texttt{ecx}_{in} = \texttt{42800000h}$$

Model for `((vEax+vEcx) == 0) != ((vEcx+vEax+vEcx+vEax) == 0)`

- If we sample the instruction's behavior in this input state, one of the hypotheses must become falsified, since:
  - If $\texttt{zf}_{out} = 1$, then `(vEax+vEcx) == 0` is false.
  - If $\texttt{zf}_{out} = 0$, then `(vEcx+vEax+vEcx+vEax) == 0` is false.
    - Hypothesis #2 is falsified.

CPU Emulator Synthesis

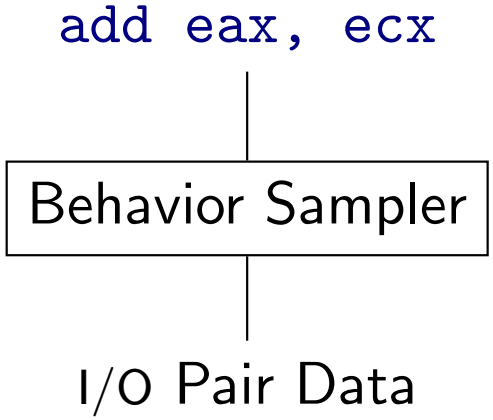
Behavior Sampler

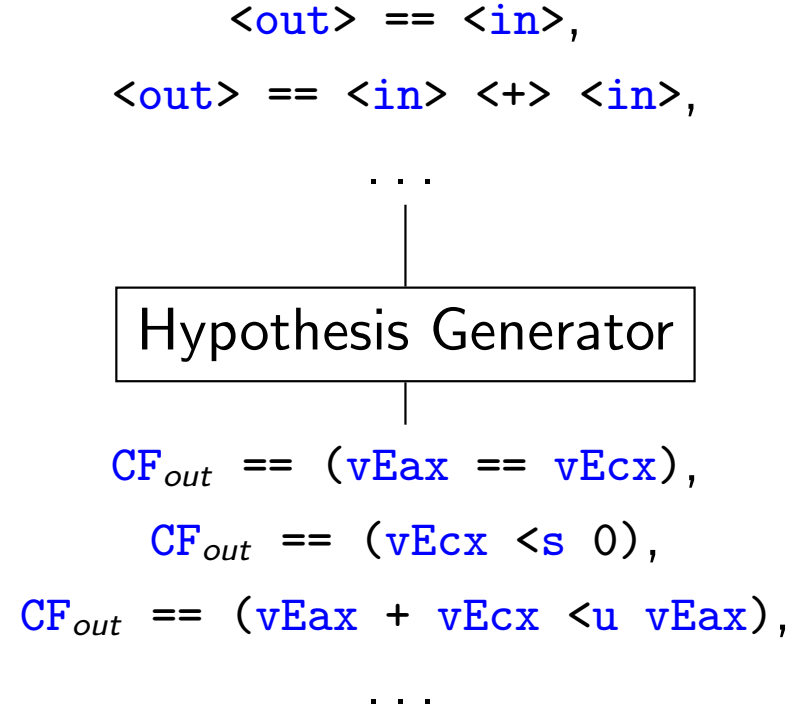


First, we generate hypotheses for the instruction's effects.

CPU Emulator Synthesis

Hypothesis Generator

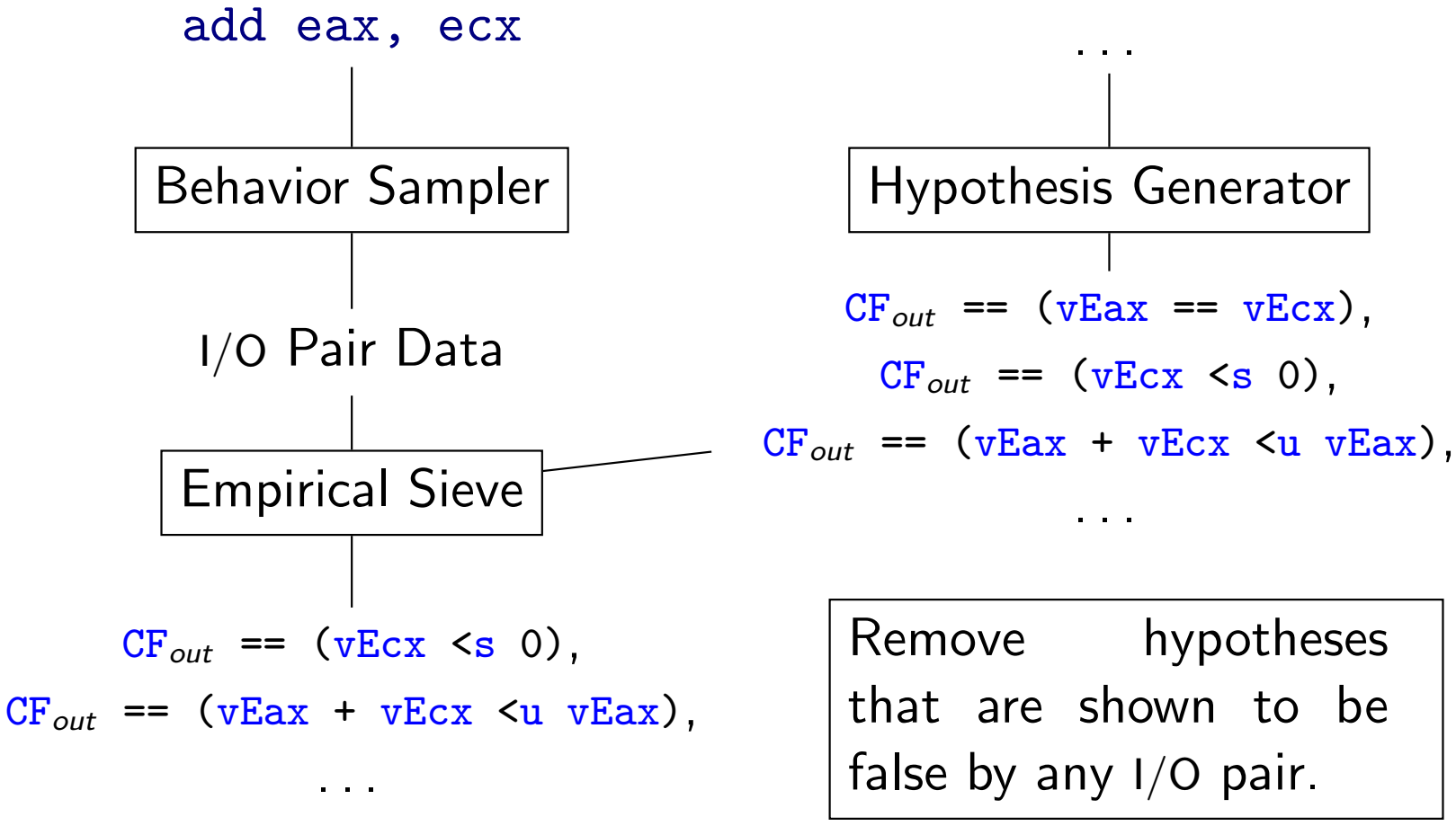




Next, we collect I/O pairs for an instruction.

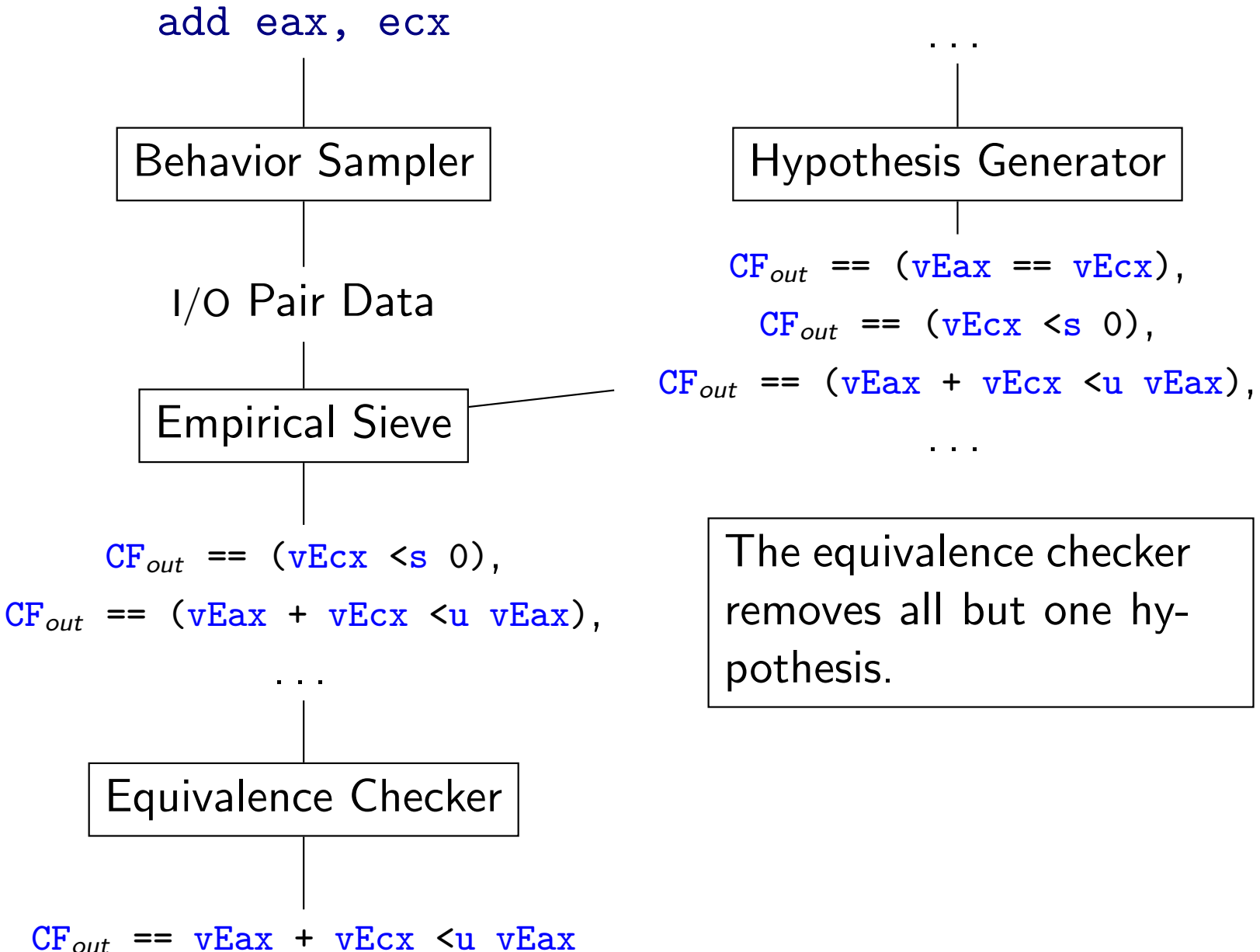
CPU Emulator Synthesis

Empirical Sieve



CPU Emulator Synthesis

Equivalence Checking



Peephole Superdeobfuscation

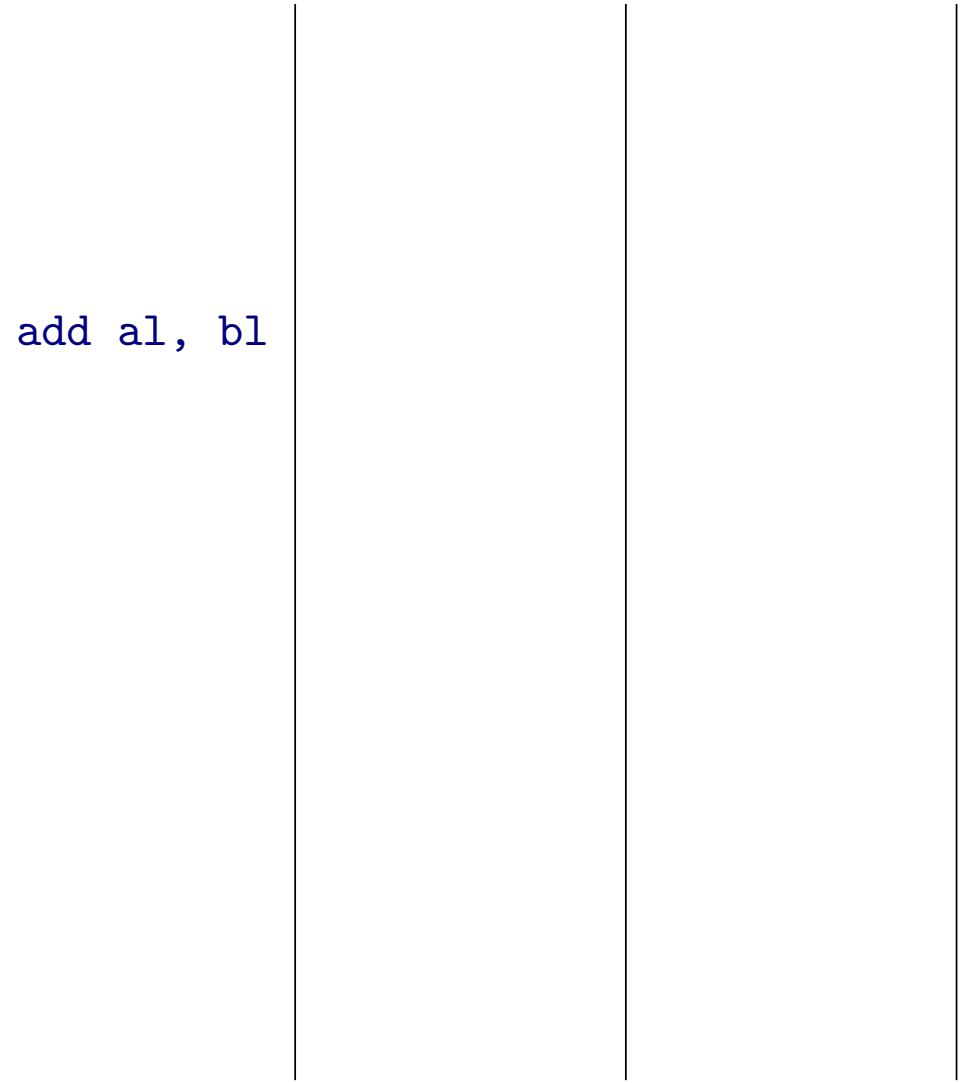
Rule-Based Obfuscation

Obfuscator Rules

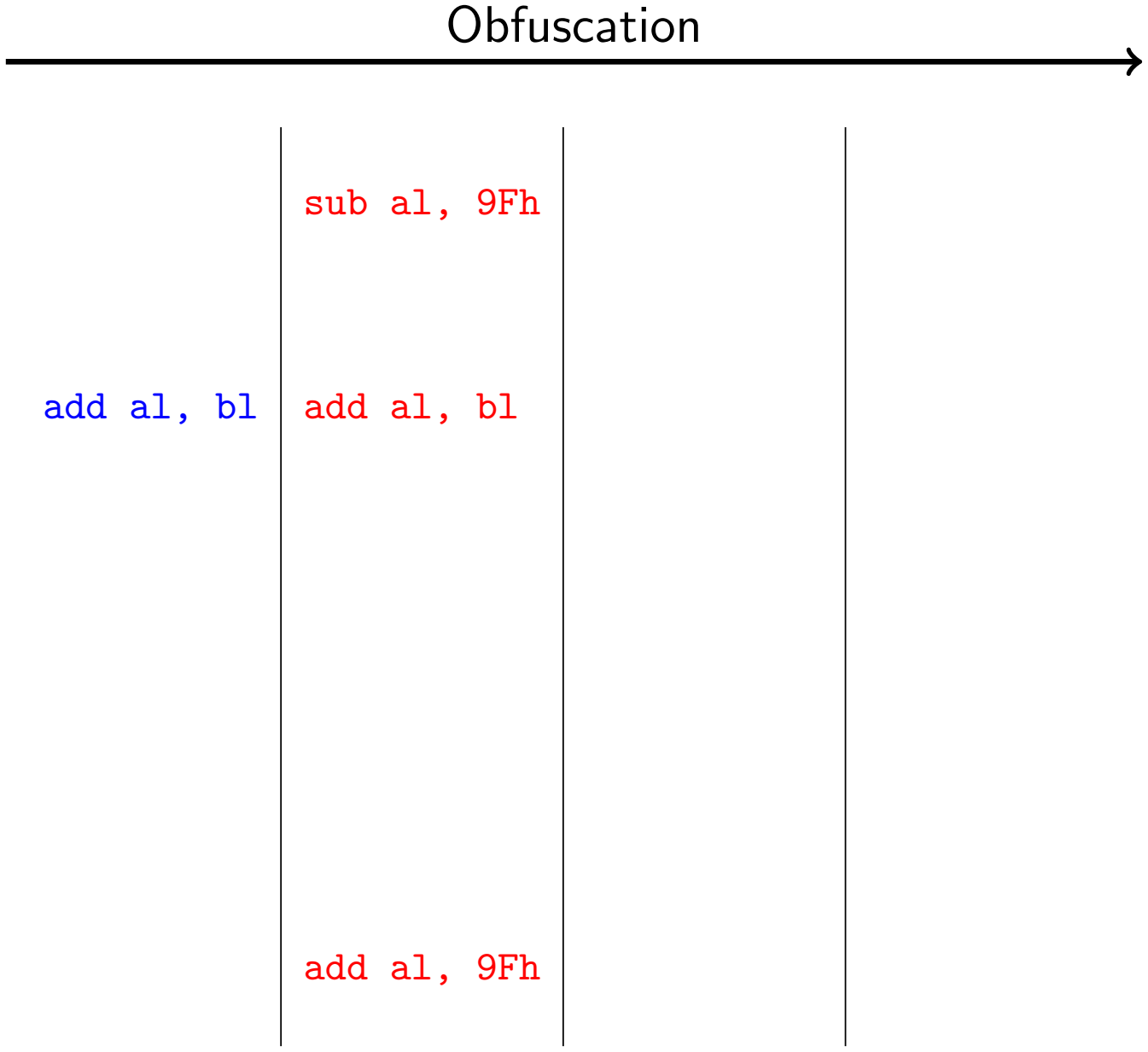
Unobfuscated	Obfuscated	Constraints
`add` $\texttt{r8}_1$`,` $\texttt{r8}_2$	`add` $\texttt{r8}_1$`,` `imm8`	
	`add` $\texttt{r8}_1$`,` $\texttt{r8}_2$	
	`sub` $\texttt{r8}_1$`,` `imm8`	
`sub` $\texttt{r32}_1$`,` $\texttt{r32}_2$	`push` $\texttt{r32}_3$	$\texttt{r32}_3 \neq \texttt{r32}_1$
	`mov` $\texttt{r32}_3$`,` $\texttt{r32}_2$	$\texttt{r32}_1 \neq$ `esp`
	`sub` $\texttt{r32}_1$`,` $\texttt{r32}_3$	$\texttt{r32}_1 \neq$ `esp`
	`pop` $\texttt{r32}_3$	$\texttt{r32}_3 \neq$ `esp`

- Some obfuscators use a rule set to randomly permute code.
- The ordinary and obfuscated sequences must have the same (or at least "similar") effects when executed.

Peephole Superdeobfuscation

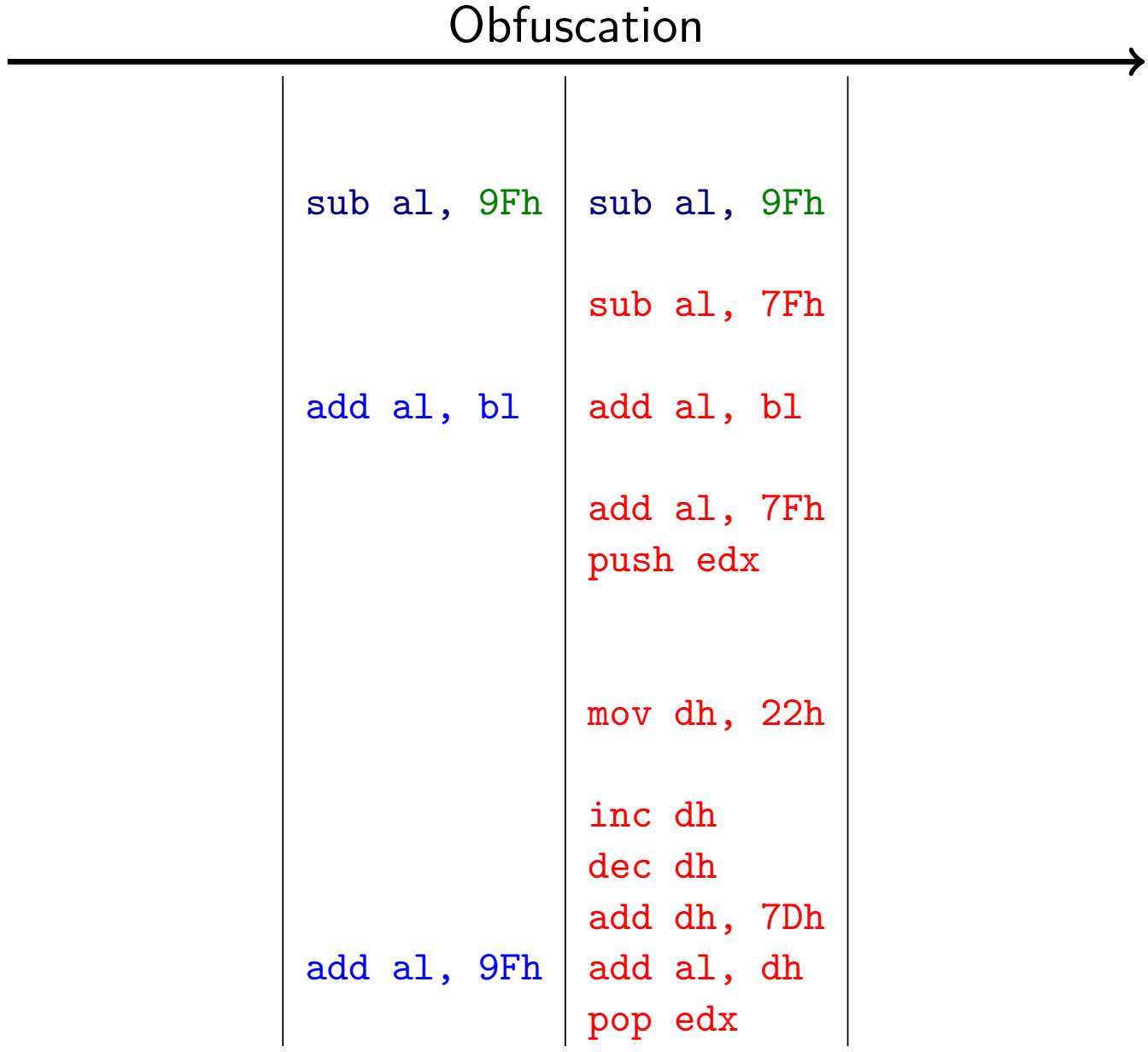


Peephole Superdeobfuscation



Replace `instructions` with `obfuscated sequences`

Peephole Superdeobfuscation



Replace instructions with obfuscated sequences

Peephole Superdeobfuscation

			`push cx`
			`mov ch, 9Fh`
		`sub al, 9Fh`	`sub al, ch`
			`pop cx`
		`sub al, 7Fh`	`sub al, 7Fh`
			`sub al, 70h`
		`add al, bl`	`add al, bl`
			`add al, 70h`
		`add al, 7Fh`	`add al, 7Fh`
		`push edx`	`push edx`
			`push ecx`
			`mov ch, 22h`
		`mov dh, 22h`	`mov dh, ch`
			`pop ecx`
		`inc dh`	`inc dh`
		`dec dh`	`dec dh`
		`add dh, 7Dh`	`add dh, 7Dh`
		`add al, dh`	`add al, dh`
		`pop edx`	`pop edx`

Replace `instructions` with `obfuscated sequences`

Peephole Superdeobfuscation

```
push cx
mov ch, 9Fh
sub al, ch
pop cx
sub al, 7Fh
sub al, 70h
add al, bl
add al, 70h
add al, 7Fh
push edx
push ecx
mov ch, 22h
mov dh, ch
pop ecx
inc dh
dec dh
add dh, 7Dh
add al, dh
pop edx
```

Peephole Superdeobfuscation

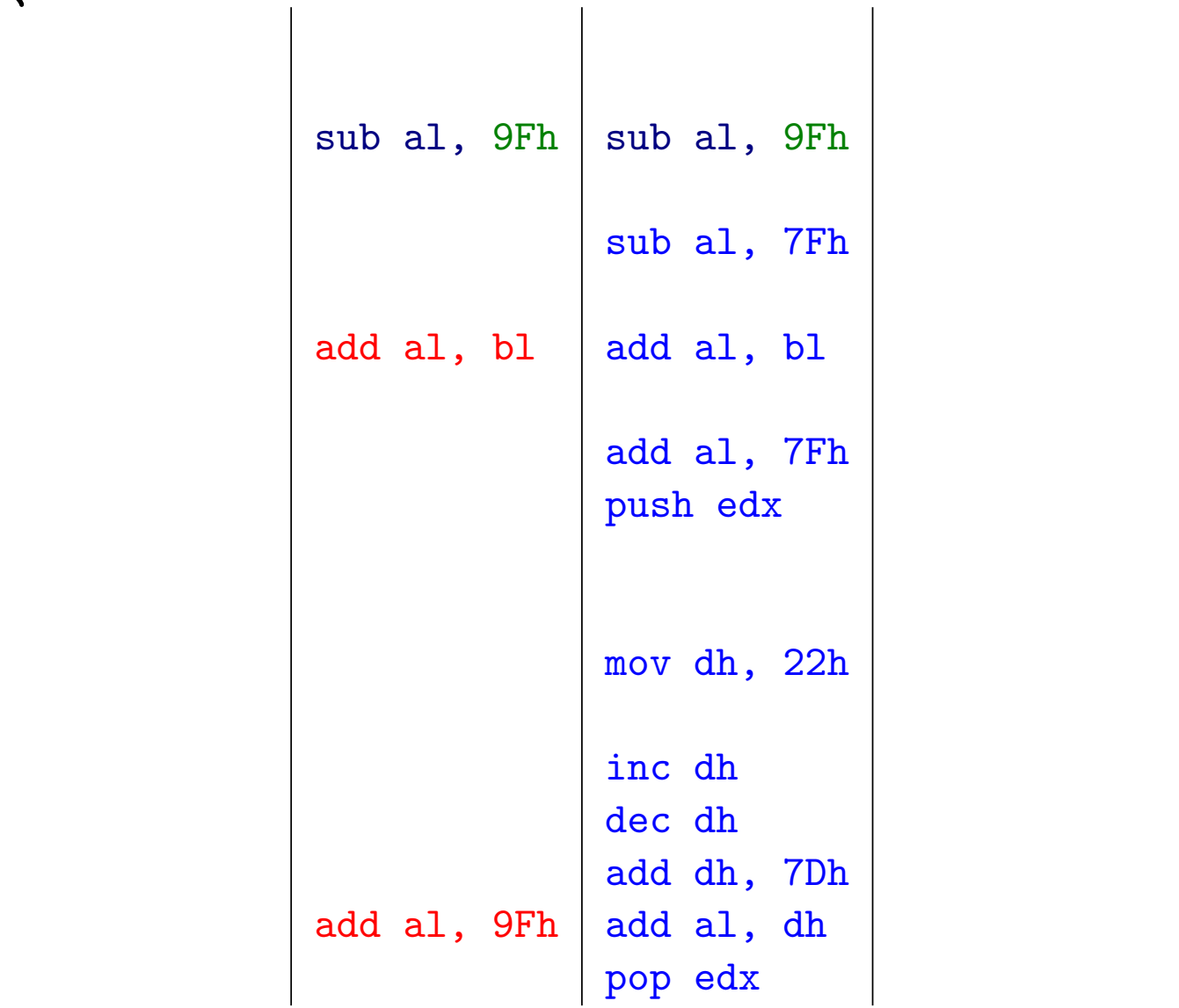
Deobfuscation with Inverse Rules

		push cx
		mov ch, 9Fh
	sub al, 9Fh	sub al, ch
		pop cx
	sub al, 7Fh	sub al, 7Fh
		sub al, 70h
	add al, bl	add al, bl
		add al, 70h
	add al, 7Fh	add al, 7Fh
	push edx	push edx
		push ecx
		mov ch, 22h
	mov dh, 22h	mov dh, ch
		pop ecx
	inc dh	inc dh
	dec dh	dec dh
	add dh, 7Dh	add dh, 7Dh
	add al, dh	add al, dh
	pop edx	pop edx

Replace obfuscated sequences with instructions

Peephole Superdeobfuscation

Deobfuscation with Inverse Rules

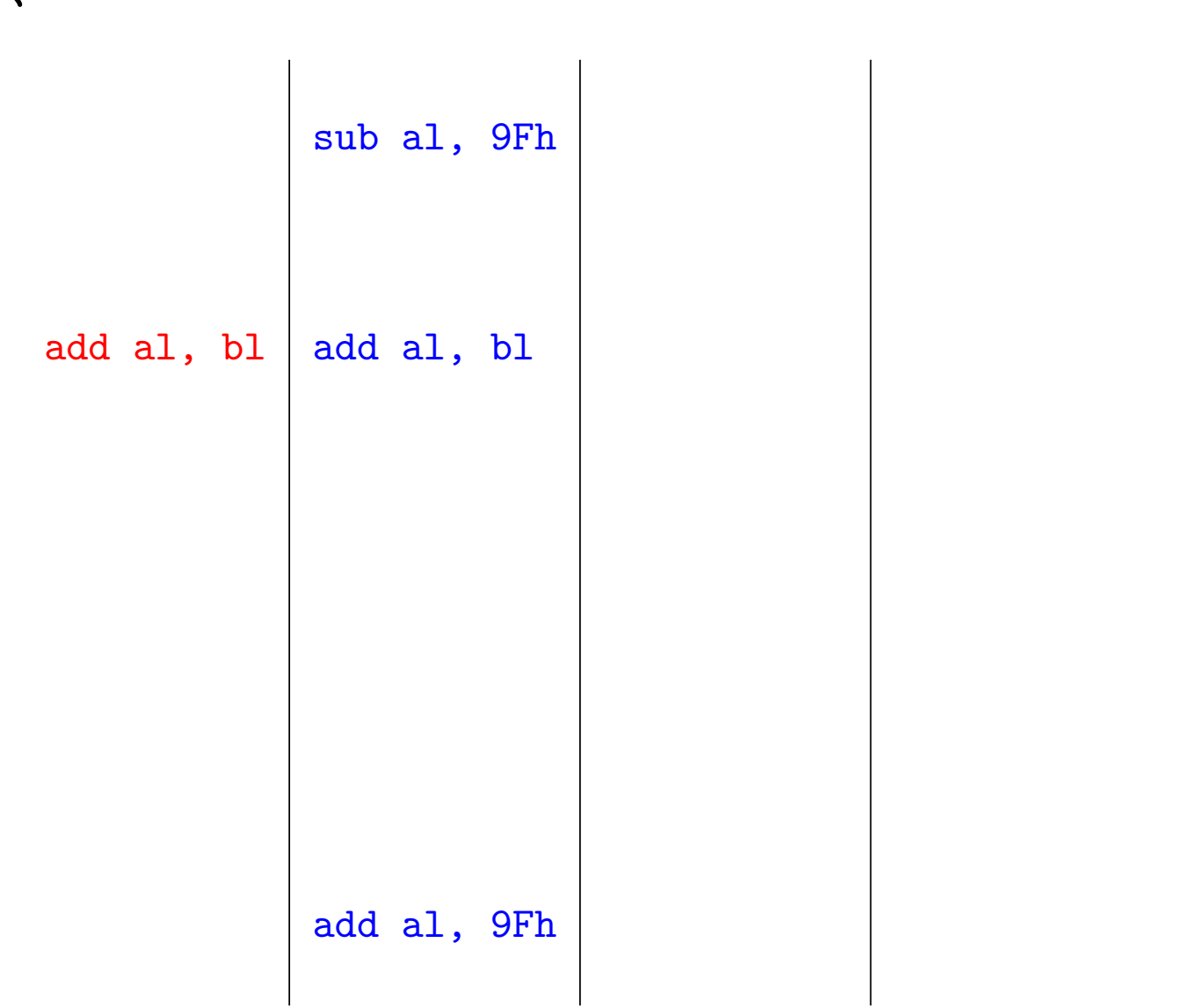


`sub al, 9Fh`	`sub al, 9Fh`
	`sub al, 7Fh`
`add al, bl`	`add al, bl`
	`add al, 7Fh`
	`push edx`
	`mov dh, 22h`
	`inc dh`
	`dec dh`
	`add dh, 7Dh`
`add al, 9Fh`	`add al, dh`
	`pop edx`

Replace `obfuscated sequences` with `instructions`

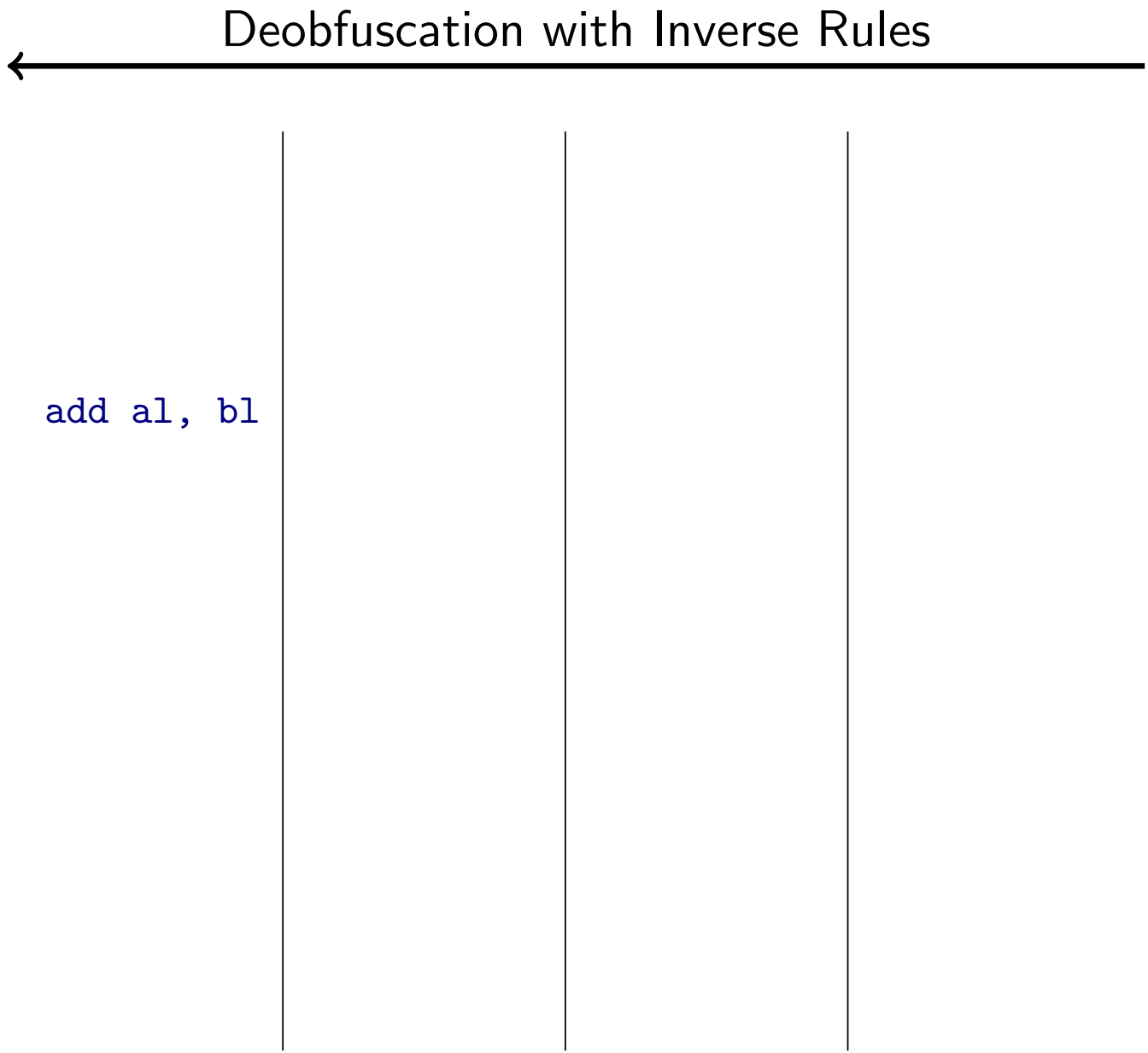
Peephole Superdeobfuscation

Deobfuscation with Inverse Rules



Replace obfuscated sequences with instructions

Peephole Superdeobfuscation



Peephole Superdeobfuscation

Rule-Based Deobfuscation

Deobfuscator Rules

Obfuscated	Unobfuscated	Constraints
`add` $\texttt{r8}_1$`,` `imm8`	`add` $\texttt{r8}_1$`,` $\texttt{r8}_2$	
`add` $\texttt{r8}_1$`,` $\texttt{r8}_2$		
`sub` $\texttt{r8}_1$`,` `imm8`		
`push` $\texttt{r32}_3$	`sub` $\texttt{r32}_1$`,` $\texttt{r32}_2$	$\texttt{r32}_3 \neq \texttt{r32}_1$
`mov` $\texttt{r32}_3$`,` $\texttt{r32}_2$		$\texttt{r32}_1 \neq \texttt{esp}$
`sub` $\texttt{r32}_1$`,` $\texttt{r32}_3$		$\texttt{r32}_1 \neq \texttt{esp}$
`pop` $\texttt{r32}_3$		$\texttt{r32}_3 \neq \texttt{esp}$

- We could inspect the code manually, pick out patterns, and write deobfuscator rules.
- This is Tedious And Error-Prone™.
- We automate the process with program synthesis.

Peephole Superdeobfuscation

The Idea: Obfuscated/Unobfuscated Behavior Must Match

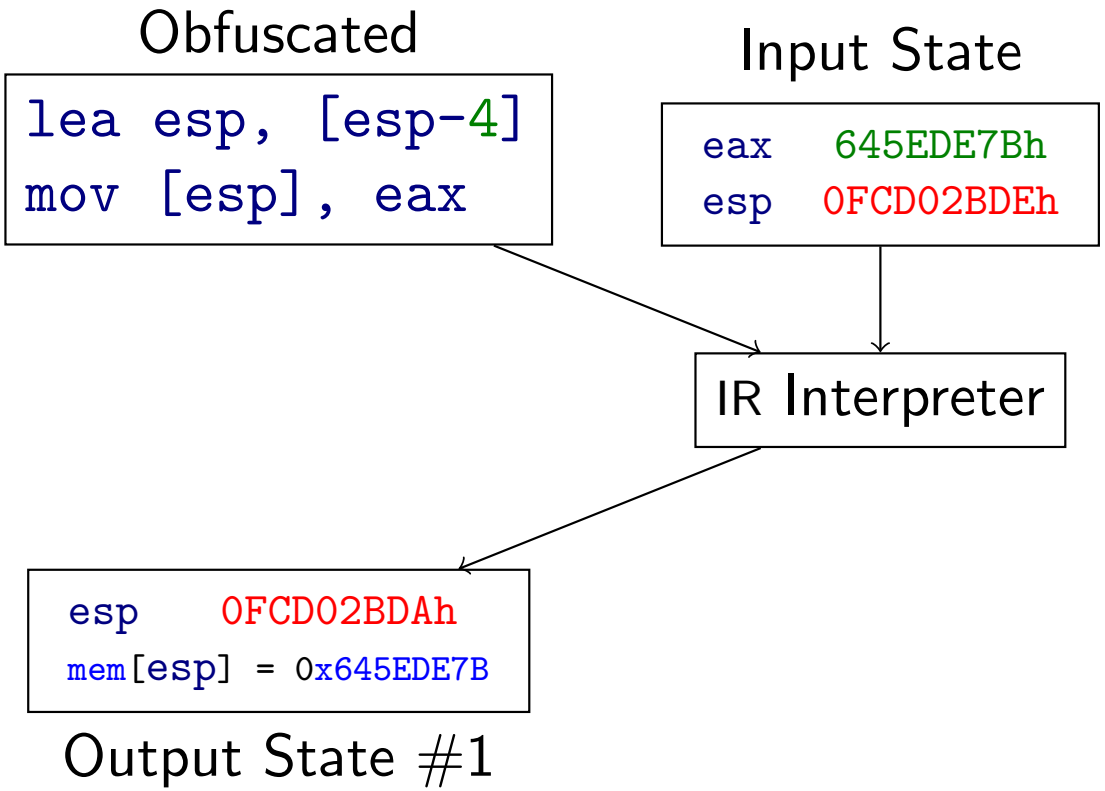
Obfuscated

```
lea esp, [esp-4]
mov [esp], eax
```

Collect obfuscated sequences.

Peephole Superdeobfuscation

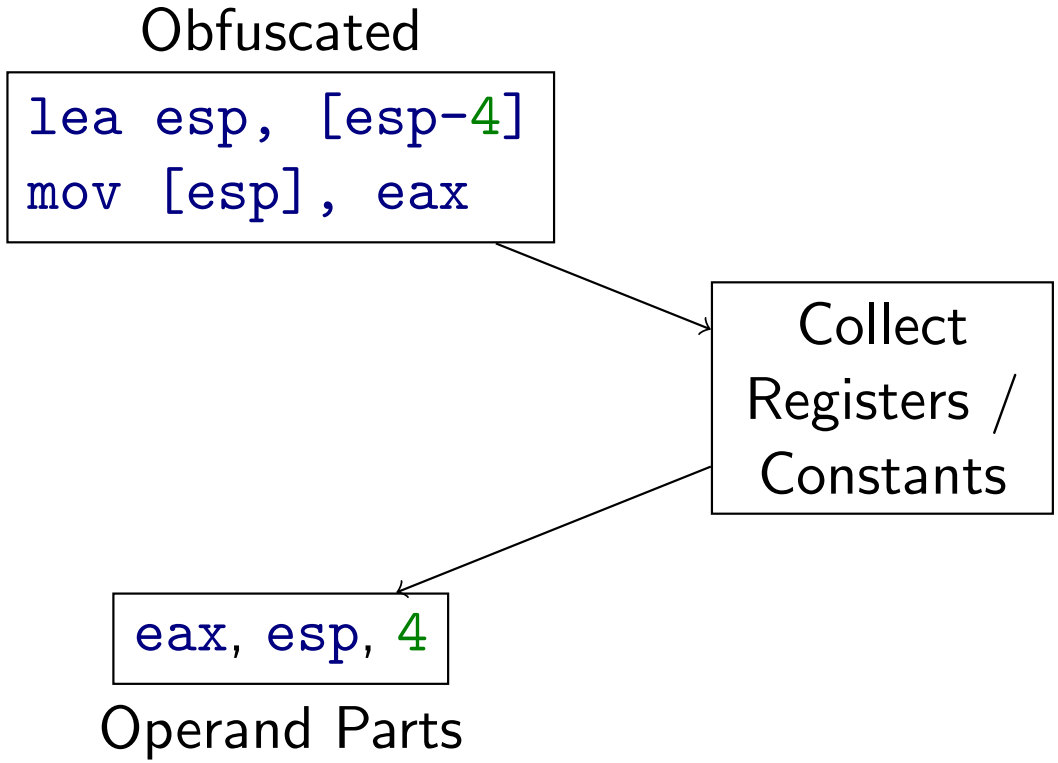
The Idea: Obfuscated/Unobfuscated Behavior Must Match



Generate I/O pairs.

Peephole Superdeobfuscation

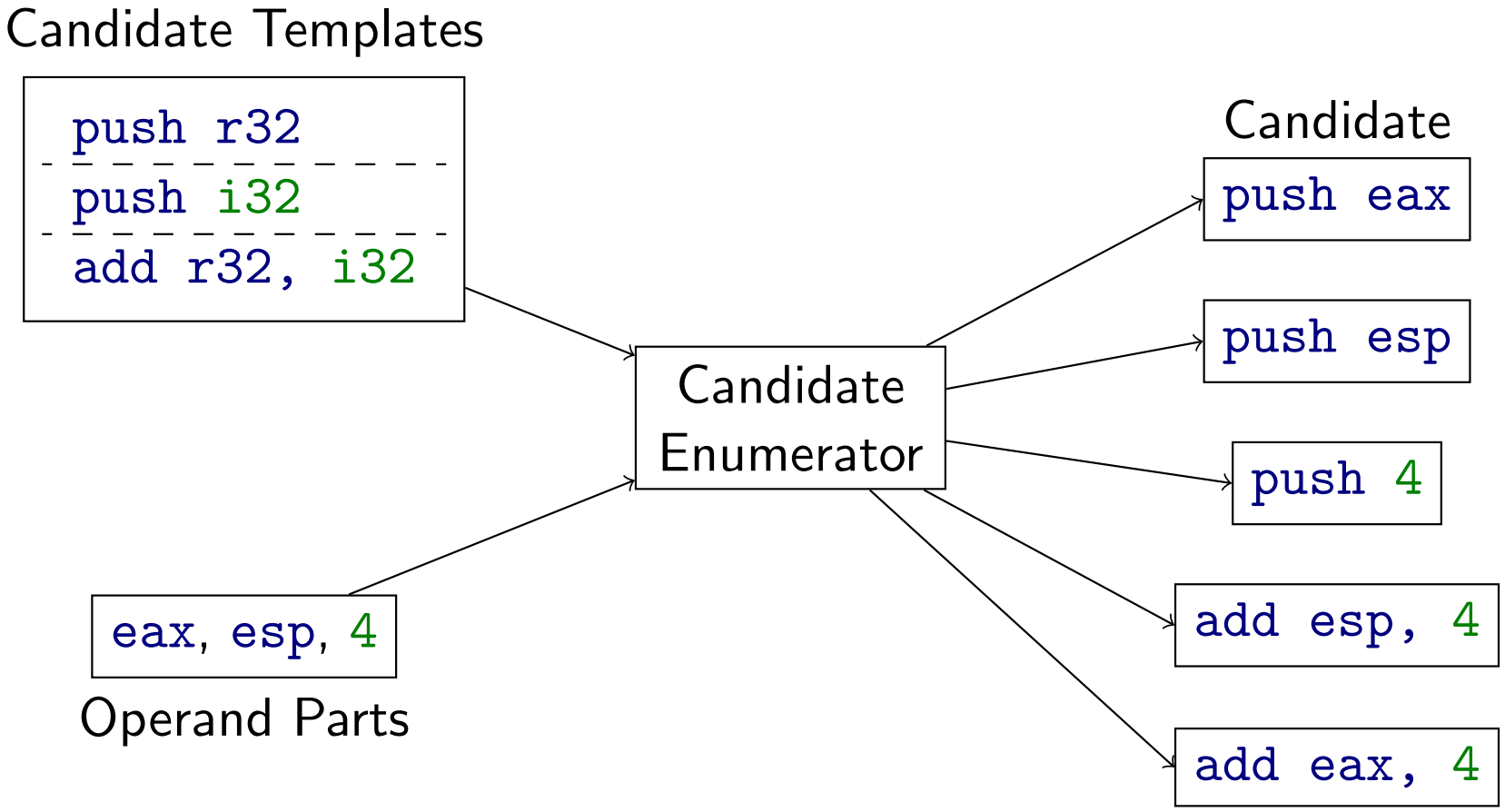
The Idea: Obfuscated/Unobfuscated Behavior Must Match



Preparing for candidate generation, collect registers and constants from the obfuscated sequence.

Peephole Superdeobfuscation

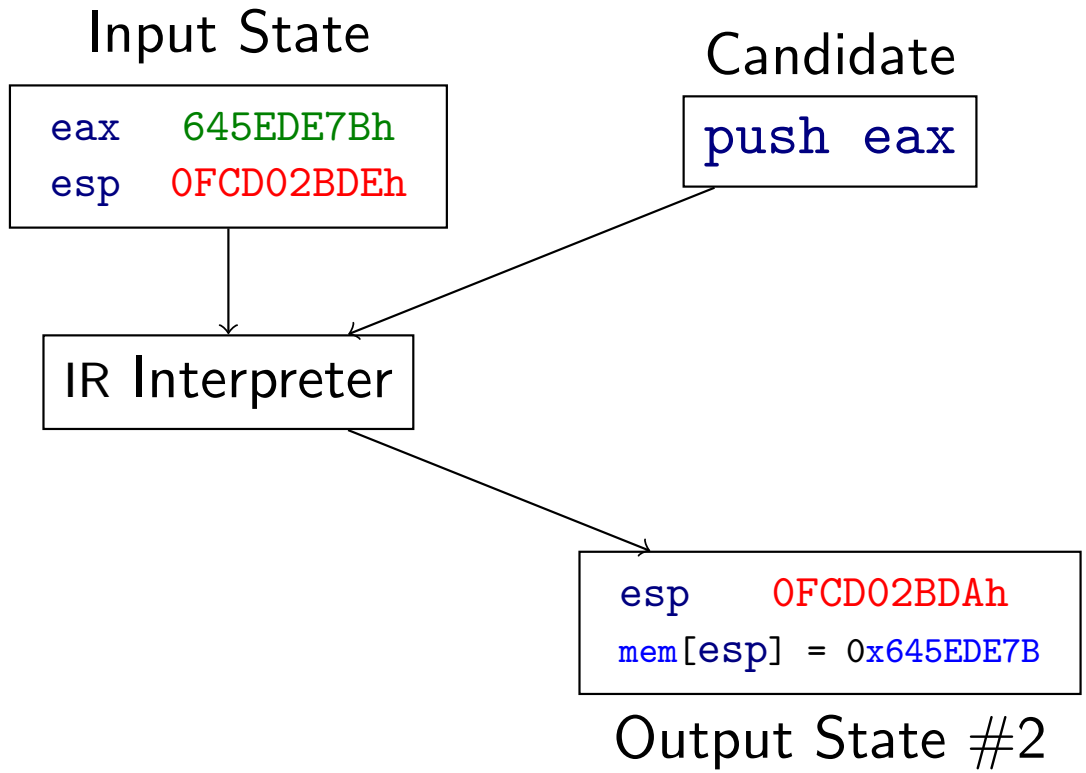
The Idea: Obfuscated/Unobfuscated Behavior Must Match



Plug the registers and constants into the templates to generate candidate replacements.

Peephole Superdeobfuscation

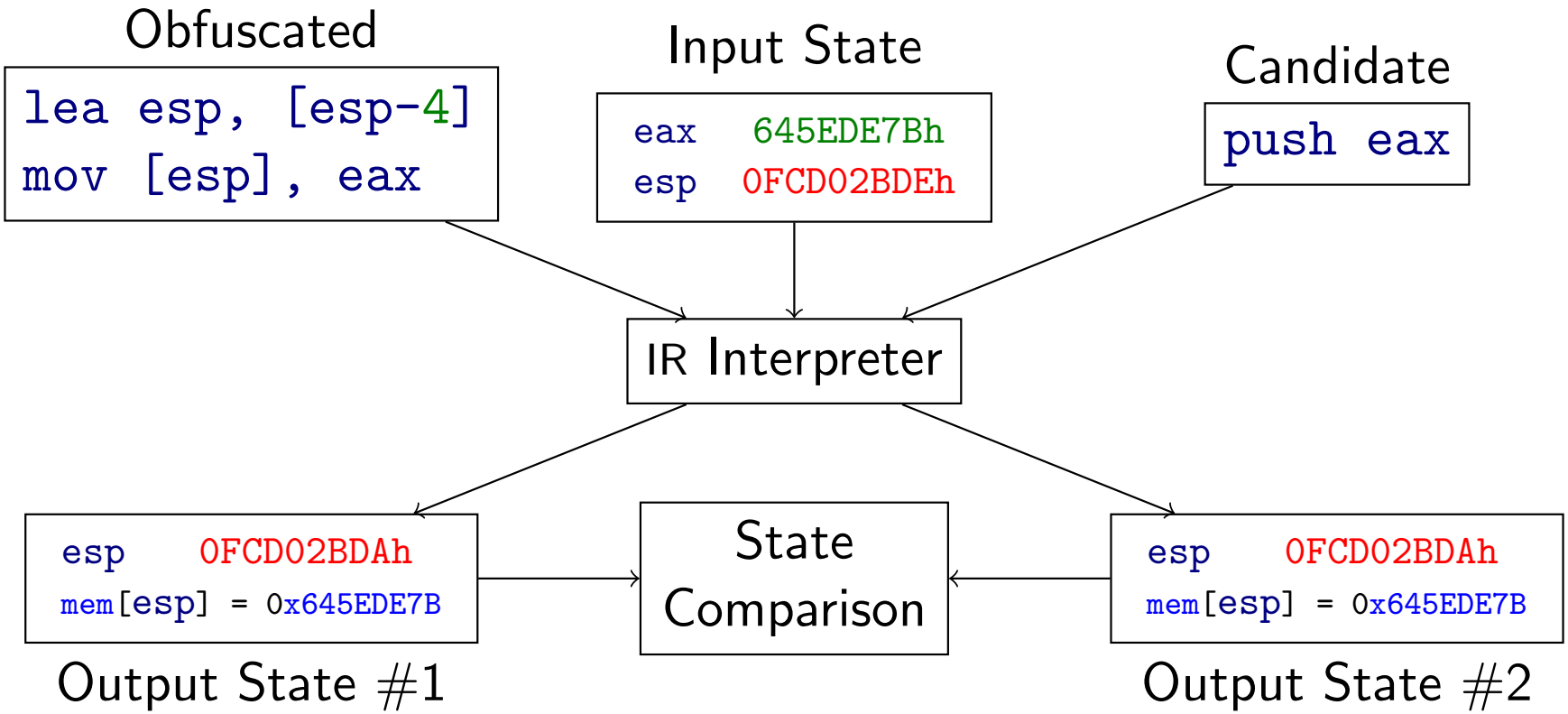
The Idea: Obfuscated/Unobfuscated Behavior Must Match



For each candidate, generate I/O pairs **in the same input states as for the original sequence**.

Peephole Superdeobfuscation

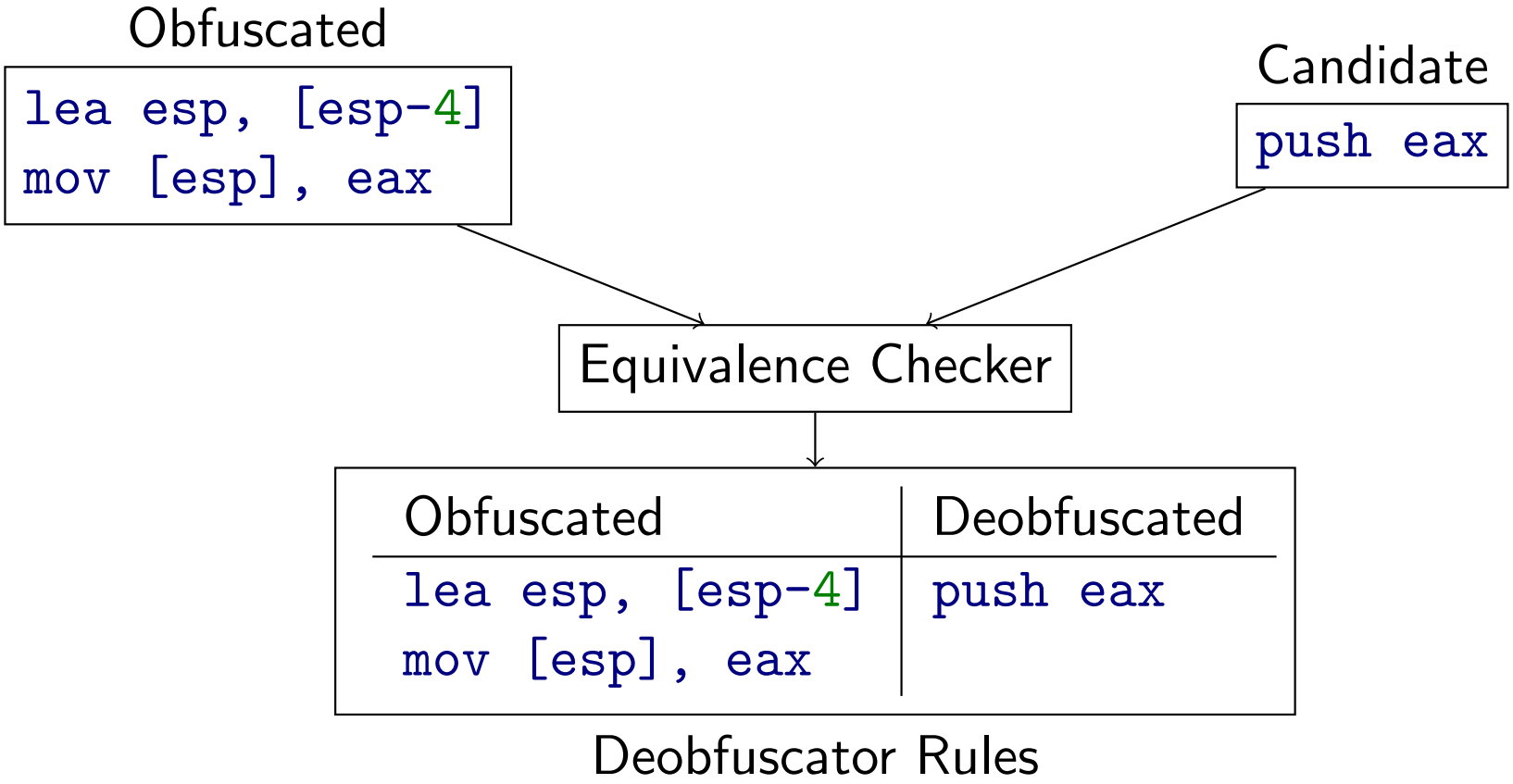
The Idea: Obfuscated/Unobfuscated Behavior Must Match



Compare the resulting states. We can be lenient for flags and stack memory. If they match, the candidate is a potential replacement for the obfuscated sequence.

Peephole Superdeobfuscation

The Idea: Obfuscated/Unobfuscated Behavior Must Match



If the states matched, use an SMT solver to ensure that the sequences are actually equivalent. Learn rules if so.

Peephole Superdeobfuscation

Generalization

Deobfuscator Rules

Obfuscated	Deobfuscated	Constraints
`lea esp, [esp-4]` `mov [esp], eax`	`push eax`	

- Our deobfuscator rule is specific to the register `eax`.
- In fact, `eax` can be replaced with any register except `esp`.

Peephole Superdeobfuscation

Generalization

Deobfuscator Rules

Obfuscated	Deobfuscated	Constraints
`lea esp, [esp-4]` `mov [esp], eax`	`push eax`	

Generalized Rules

`lea esp, [esp-4]` `mov [esp], r32`	`push r32`	`r32` $\neq$ `esp`

- Our deobfuscator rule is specific to the register `eax`.
- In fact, `eax` can be replaced with any register except `esp`.
- **Generalization** removes specific register and/or immediate values from deobfuscation rules.

Peephole Superdeobfuscation

In Action

```
▶  sub esp, 4
   mov [esp], eax
   mov eax, ebp
   push ebp
   push edi
   mov [esp], eax
   xor [esp], 7FEE03B1h
   pop ebp
   sub esp, 4
   mov [esp], esi
   push edx
   mov edx, 6F6B5081h
   mov esi, 0BC6D38Bh
   add esi, edx
   pop edx
```

Obfuscated	Deobfuscated	Constraints

We iterate through the instructions, trying to simplify those within a window.

▶: bottom of current window.

▶: can simplify.

▶: simplification applied.

Peephole Superdeobfuscation

In Action

```
sub esp, 4
mov [esp], eax
mov eax, ebp
push ebp
push edi
mov [esp], eax
xor [esp], 7FEE03B1h
pop ebp
sub esp, 4
mov [esp], esi
push edx
mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$

Learn rule and generalize.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp
push edi
mov [esp], eax
xor [esp], 7FEE03B1h
pop ebp
sub esp, 4
mov [esp], esi
push edx
mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	`r32` $\neq$ `esp`

No simplification.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp
push edi
mov [esp], eax
xor [esp], 7FEE03B1h
pop ebp
sub esp, 4
mov [esp], esi
push edx
mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	`r32` $\neq$ `esp`

No simplification.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp
▶ push edi
mov [esp], eax
xor [esp], 7FEE03B1h
pop ebp
sub esp, 4
mov [esp], esi
push edx
mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	`r32` $\neq$ `esp`

No simplification.

Peephole Superdeobfuscation

In Action

```
    push eax

    mov eax, ebp
    push ebp
▶   push edi
▶   mov [esp], eax
    xor [esp], 7FEE03B1h
    pop ebp
    sub esp, 4
    mov [esp], esi
    push edx
    mov edx, 6F6B5081h
    mov esi, 0BC6D38Bh
    add esi, edx
    pop edx
```

	Obfuscated	Deobfuscated	Constraints
	`sub esp, 4` `mov [esp], r32`	`push r32`	$\texttt{r32} \neq \texttt{esp}$
▶ ▶	`push` $\texttt{r32}_1$ `mov [esp],` $\texttt{r32}_2$	`push` $\texttt{r32}_2$	$\texttt{r32}_2 \neq \texttt{esp}$

Learn rule and generalize.

Peephole Superdeobfuscation

In Action

```
    push eax

    mov eax, ebp
    push ebp
▶   push eax

▶   xor [esp], 7FEE03B1h
    pop ebp
    sub esp, 4
    mov [esp], esi
    push edx
    mov edx, 6F6B5081h
    mov esi, 0BC6D38Bh
    add esi, edx
    pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	$\texttt{r32} \neq \texttt{esp}$
`push` $\texttt{r32}_1$ `mov [esp],` $\texttt{r32}_2$	`push` $\texttt{r32}_2$	$\texttt{r32}_2 \neq \texttt{esp}$

No simplification.

Peephole Superdeobfuscation

In Action

```
    push eax

    mov eax, ebp
    push ebp
▶   push eax

▶   xor [esp], 7FEE03B1h
▶   pop ebp
    sub esp, 4
    mov [esp], esi
    push edx
    mov edx, 6F6B5081h
    mov esi, 0BC6D38Bh
    add esi, edx
    pop edx
```

	Obfuscated	Deobfuscated	Constraints
	`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
	`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
▶ ▶ ▶	`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$

Learn rule and generalize.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp
```

- `mov ebp, eax`
- `xor ebp, 7FEE03B1h`
- `sub esp, 4`

```
mov [esp], esi
push edx
mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$

No simplification.

Peephole Superdeobfuscation

In Action

```
    push eax

    mov eax, ebp
    push ebp

    mov ebp, eax
    xor ebp, 7FEE03B1h
▶   sub esp, 4
▶   mov [esp], esi
    push edx
    mov edx, 6F6B5081h
    mov esi, 0BC6D38Bh
    add esi, edx
    pop edx
```

	Obfuscated	Deobfuscated	Constraints
▶	`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
	`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
	`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$

Apply previous simplification.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp


mov ebp, eax
xor ebp, 7FEE03B1h
push esi

push edx
mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$

No simplification.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp


mov ebp, eax
xor ebp, 7FEE03B1h
push esi

push edx
▶ mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$

No simplification.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp


mov ebp, eax
xor ebp, 7FEE03B1h
push esi

push edx
mov edx, 6F6B5081h
▶ mov esi, 0BC6D38Bh
add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$

No simplification.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp


mov ebp, eax
xor ebp, 7FEE03B1h
push esi

push edx
mov edx, 6F6B5081h
mov esi, 0BC6D38Bh
▶ add esi, edx
pop edx
```

Obfuscated	Deobfuscated	Constraints
`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$

No simplification.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp

mov ebp, eax
xor ebp, 7FEE03B1h
push esi
```

- `push edx`
- `mov edx, 6F6B5081h`
- `mov esi, 0BC6D38Bh`
- `add esi, edx`
- `pop edx`

	Obfuscated	Deobfuscated	Constraints
	`sub esp, 4` `mov [esp], r32`	`push r32`	$r32 \neq esp$
	`push` $r32_1$ `mov [esp],` $r32_2$	`push` $r32_2$	$r32_2 \neq esp$
	`push` $r32_1$ `xor [esp], i32` `pop` $r32_2$	`mov` $r32_2$`,` $r32_1$ `xor` $r32_2$`, i32`	$r32_1 \neq esp$ $r32_2 \neq esp$
▶	`push` $r32_1$ ▶ `mov` $r32_1$`,` $i32_1$ ▶ `mov` $r32_2$`,` $i32_2$ ▶ `add` $r32_2$`,` $r32_1$ ▶ `pop` $r32_1$	`mov` $r32_2$`,` $i32_1$`+`$i32_2$	$r32_1 \neq esp$ $r32_2 \neq esp$ $r32_1 \neq r32_2$

Learn rule and generalize.

Peephole Superdeobfuscation

In Action

```
push eax

mov eax, ebp
push ebp


mov ebp, eax
xor ebp, 7FEE03B1h
push esi

mov esi, 7B32240Ch
```

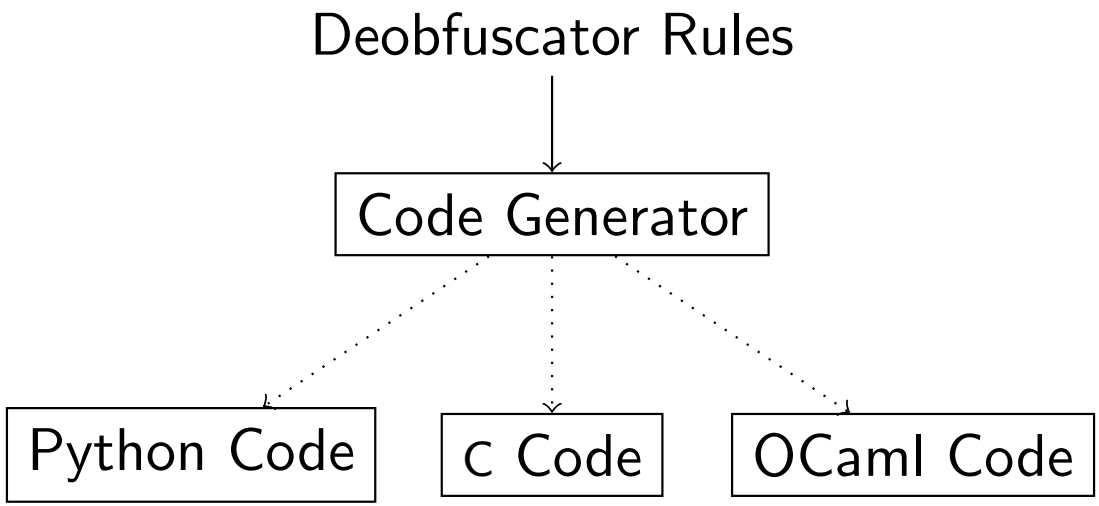
Obfuscated	Deobfuscated	Constraints
sub esp, 4 mov [esp], r32	push r32	r32 $\neq$ esp
push $r32_1$ mov [esp], $r32_2$	push $r32_2$	$r32_2 \neq$ esp
push $r32_1$ xor [esp], i32 pop $r32_2$	mov $r32_2$, $r32_1$ xor $r32_2$, i32	$r32_1 \neq$ esp $r32_2 \neq$ esp

push $r32_1$
mov $r32_1$, $i32_1$
mov $r32_2$, $i32_2$
add $r32_2$, $r32_1$
pop $r32_1$
	mov $r32_2$, $i32_1$+$i32_2$	$r32_1 \neq$ esp
		$r32_2 \neq$ esp
		$r32_1 \neq r32_2$

Continue ad infinitum.

Peephole Superdeobfuscation

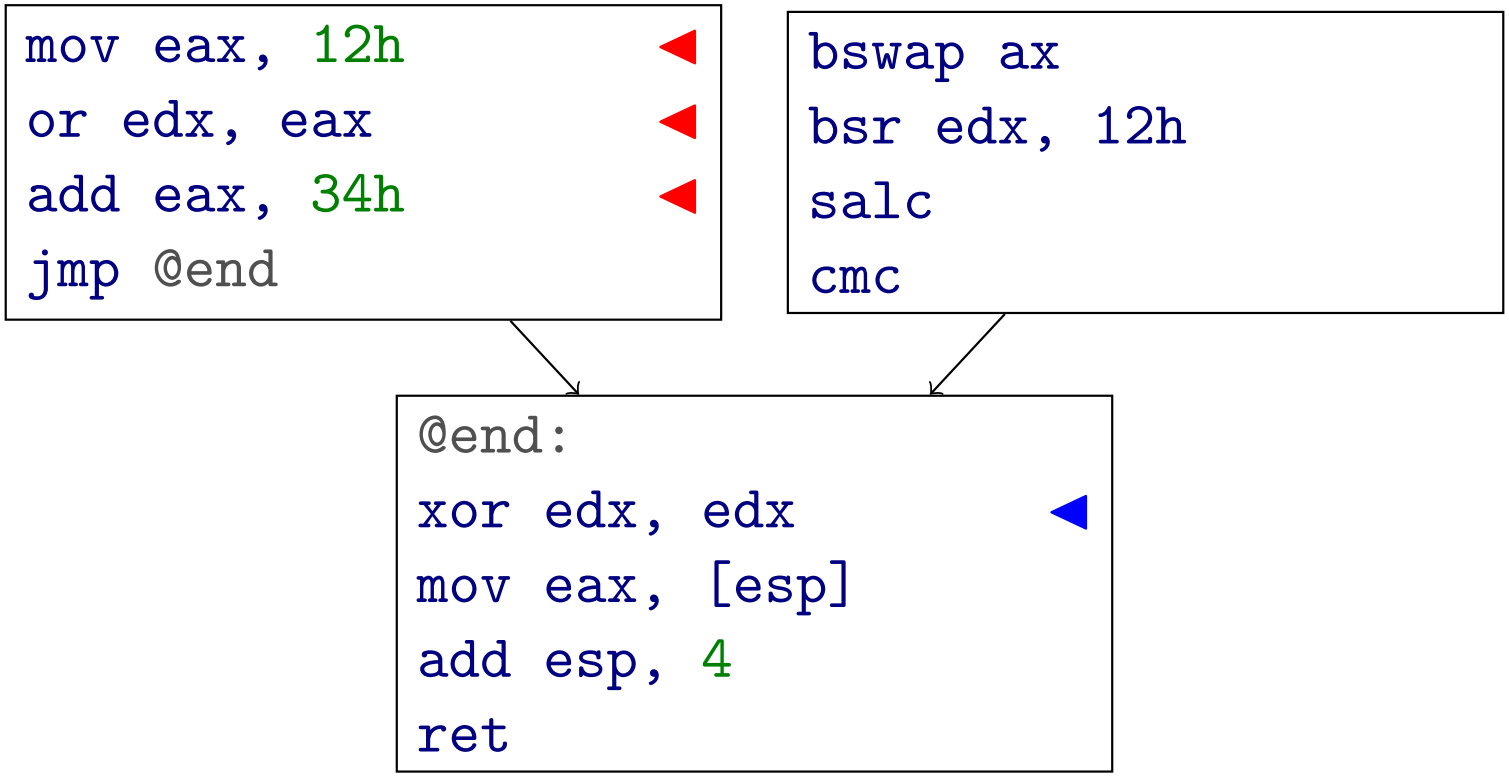
System Output



- We can turn a set of rules into a program that uses pattern-matching to implement those transformations.
- We can generate a deobfuscator automatically, and also the obfuscator that created the code in the first place!

Peephole Superdeobfuscation

Limitations



- Could simplify ◀ if we knew that `edx` was **dead** (i.e., not used before its next modification ◀).
- Ours is a forward analysis; dead code elimination requires backwards analysis.
  - Can incorporate a liveness analysis for this purpose.

Template-Based Program Synthesis

Overview

Question: is it possible to create the function `x+1` by using two of `~` (`not`) and/or `-` (`neg`)?

`y = ~x;` `z = ~y;`	`y = -x;` `z = ~y;`
`y = ~x;` `z = -y;`	`y = -x;` `z = -y;`

Table: All Possible Sequences

Template-Based Program Synthesis

Templates

Problem phrased as a **template**: what do `bop1` and `bop2` need to be set to, so that `f(x) == x+1` for all values of `x`?

```
bool bop1, bop2;

int f(int x)
{
  int y = bop1 ? -x : ~x;
  return  bop2 ? -y : ~y;
}
```

Template-Based Program Synthesis

Solving

- After phrasing the question appropriately:

English	Mathematics
Are there values of `bop1`, `bop2`	$\exists$ `bop1`, `bop2` $\in$ **Bool** $\cdot$
Such that, for all values of `x`	$\forall$ `x` $\in$ **BV**[32] $\cdot$

In the code
`y = bop1 ? -x : ~x`
`z = bop2 ? -y : ~y`
	
	let `y = bop1 ? -x : ~x` in
	let `z = bop2 ? -y : ~y` in

`z == x+1` is always `true`?	`z == x+1`

- An SMT solver gives `bop1 = false`, `bop2 = true`.
- I.e., `int f(int x) { return -~x; }`

Template-Based Program Synthesis

Extending the Framework

We can extend the idea to use more than two operator types:

```
char op1; ◀


int f(int x)
{
  y = op1 == 0 ? -x : ◀
      op1 == 1 ? ~x : ◀
                  x-1; ◀
  return y;
}
```

Or reference further constants that the solver must provide:

```
bool op1;
char c1; ◀

int f(int x)
{
  y = op1 ? x + c1 : ◀
            x ^ c1;  ◀

  return y;
}
```

Metamorphic Extraction

Metamorphic Decoder Generation

- A metamorphic engine generates code like the following:

```
lodsb
op al, bl
op al, i81
op al, i82
op bl, al
```

Where each `op` can be `add`, `sub`, or `xor`, and `i81` / `i82` are 8-bit constants. I.e., $3^4 * 2^8 * 2^8 \approx 5.3$ million possible instances.

- Example metamorphic decoder:

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

Metamorphic Extraction

Further Obfuscation

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```


```
add al, 0B7h
sub al, 90h
push cx
mov cl, bl
add al, bl
pop cx
add al, 90h
sub al, 0B7h
push ebx
mov ebx, 8716AEF1h
push ecx
push eax
xor dword ptr [esp], ebx
; continued
```

- The metamorphic code is then further obfuscated.

Metamorphic Extraction

Goal

```
add al, 0B7h
sub al, 90h
push cx
mov cl, bl
add al, bl
pop cx
add al, 90h
sub al, 0B7h
push ebx
mov ebx, 8716AEF1h
push ecx
push eax
xor dword ptr [esp], ebx
; continued
```

Re-Create →

```
lodsb
op al, bl
op al, i81
op al, i82
op bl, al
```

- Given an obfuscated sequence, we want to determine the underlying metamorphically-generated function.
  - I.e., would like to know the ops and i8s.
- We re-create the information instead of deobfuscating.
- Let's explore two approaches based on program synthesis.

Metamorphic Extraction

Template for Metamorphic Functionality

Template for Metamorphic Functionality

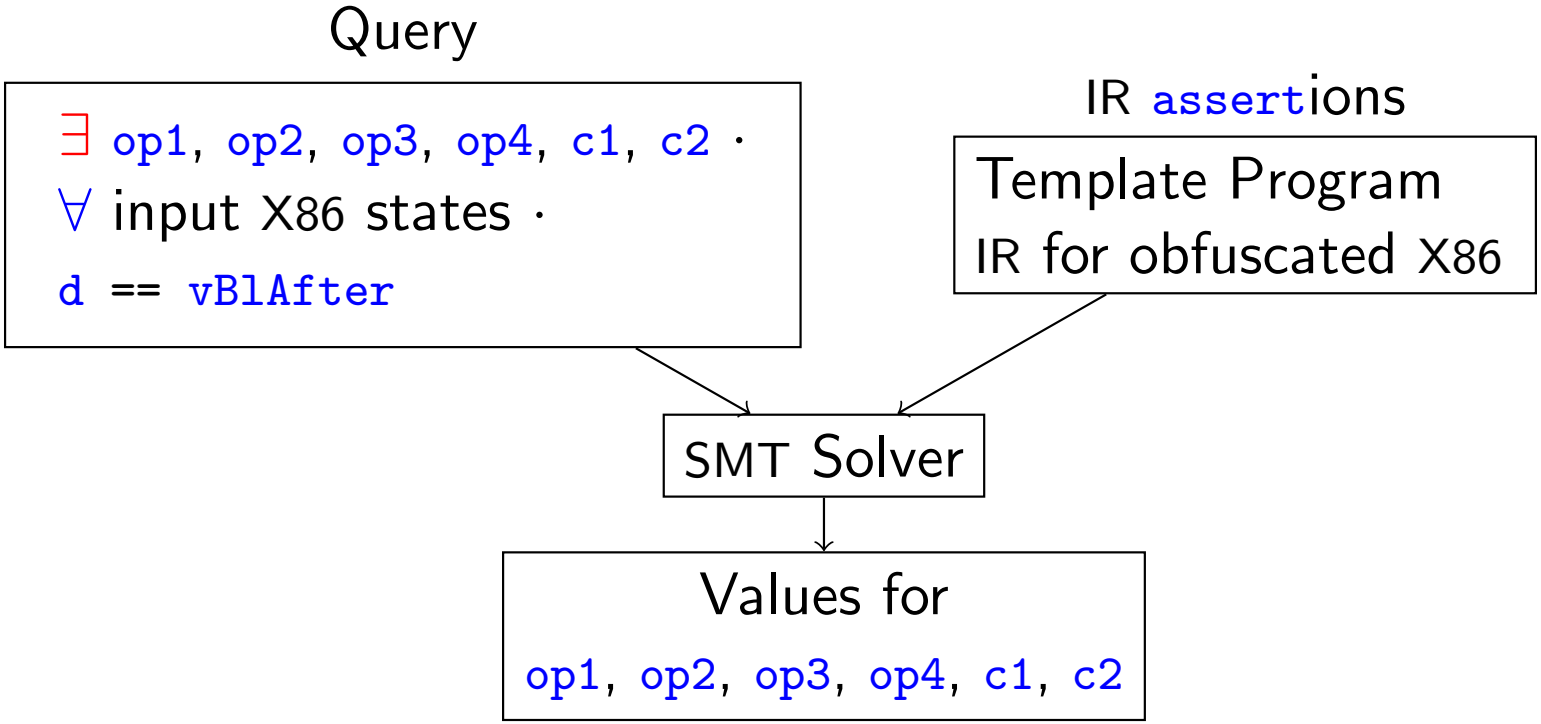
```
al =   Load(vMem,vEdi,8);                                                                    ◀
 a =   op1 == 0 ?   al + vBl   :   op1 == 1 ?   al - vBl   :   al ^ vBl;                     ◀ ◀
 b =   op2 == 0 ?    a + c1   :   op2 == 1 ?    a - c1   :    a ^ c1;                       ■
 c =   op3 == 0 ?    b + c2   :   op3 == 1 ?    b - c2   :    b ^ c2;                       ■
 d =   op4 == 0 ?   vBl + c   :   op4 == 1 ?   vBl - c   :   vBl ^ c;                       ◀ ◀
```

- X86-related variables:
  - `al` is the value read by `lodsb` ◀
  - `vBl` is the initial value of `bl` ◀
  - `d` is the final value of `bl` ◀
- Template parameters:
  - Constants `c1`, `c2` ■
  - Operators `op1`, `op2`, `op3`, `op4` ■
    - 0:add, 1:sub, 2+:xor

```
lodsb ◀
■ op al, bl ◀
■ op al, i81 ■
■ op al, i82 ■
■ op bl, al ◀
```

Metamorphic Extraction

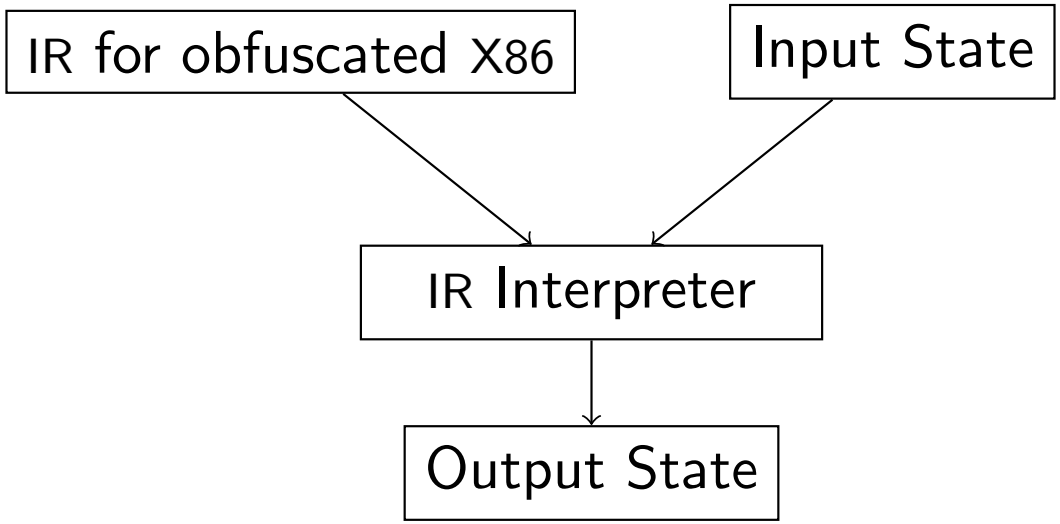
Straightforward Approach Using $\exists \cdot \forall \cdot$



- We can solve directly using the quantifiers $\exists$, $\forall$.
- Quantifiers can slow solving, so we show an alternative.

Metamorphic Extraction

Collect I/O Pairs



- Collect I/O pairs for the obfuscated X86 seqeuence.
- Extract the important parts of the states.

`al`	`0x00`	`vBl`	`0x00`	`vBlAfter`	`0xE1`

Parts of state needed for synthesis

Metamorphic Extraction

Create Witnesses

Plug the I/O pair

al	0x00	◀	vBl	0x00	◀	vBlAfter	0xE1	◀

Into the template

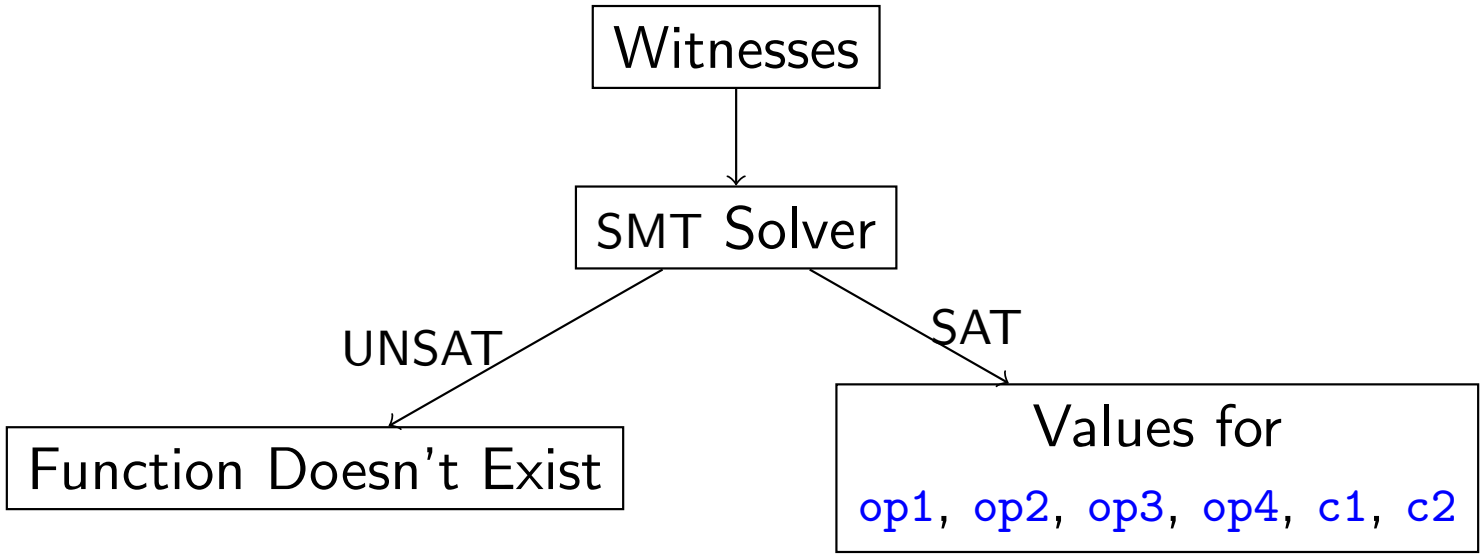
```
a = op1 == 0 ? al + vBl : op1 == 1 ? al - vBl : al ^ vBl; ◀ ◀
b = op2 == 0 ? a + c1 : op2 == 1 ? a - c1 : a ^ c1;
c = op3 == 0 ? b + c2 : op3 == 1 ? b - c2 : b ^ c2;
d = op4 == 0 ? vBl + c : op4 == 1 ? vBl - c : vBl ^ c; ◀
```

To obtain a **witness**:

```
a1 = op1 == 0 ? 0x00+0x00 : op1 == 1 ? 0x00-0x00 : 0x00^0x00; ◀ ◀
b1 = op2 == 0 ? a1 + c1 : op2 == 1 ? a1 - c1 : a1 ^ c1;
c1 = op3 == 0 ? b1 + c2 : op3 == 1 ? b1 - c2 : b1 ^ c2;
d1 = op4 == 0 ? 0x00 + c1 : 0x00 == 1 ? 0x00 - c1 : 0x00 ^ c1; ◀
assert(d1 == 0xE1); ◀
```

Metamorphic Extraction

Synthesizing Candidate Functions



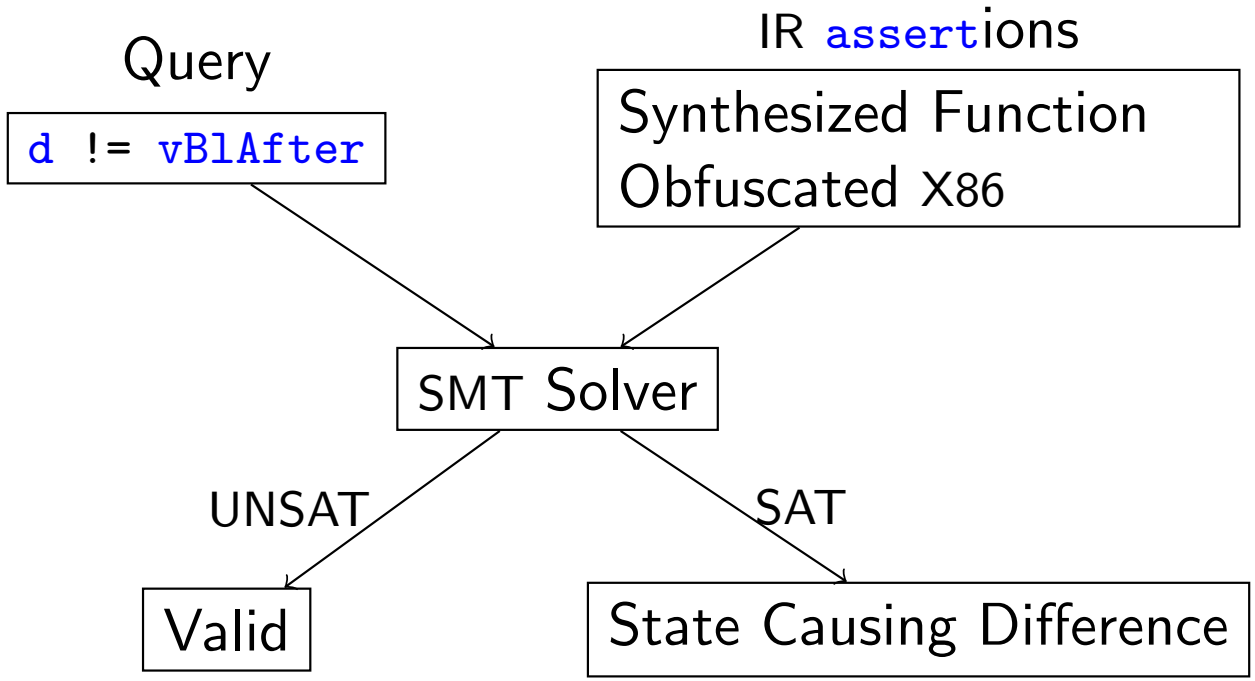
- Query for template parameters that satisfy the witnesses.
- If they exist, create a function from them.

```
a = a1 ^ vB1;
b = a ^ 0x4A;
c = b ^ 0x85;
d = vB1 + c;
```

Metamorphic Extraction

Equivalence Checking

- Our function is only valid for witnesses seen far.
- Does its behavior always match the X86?



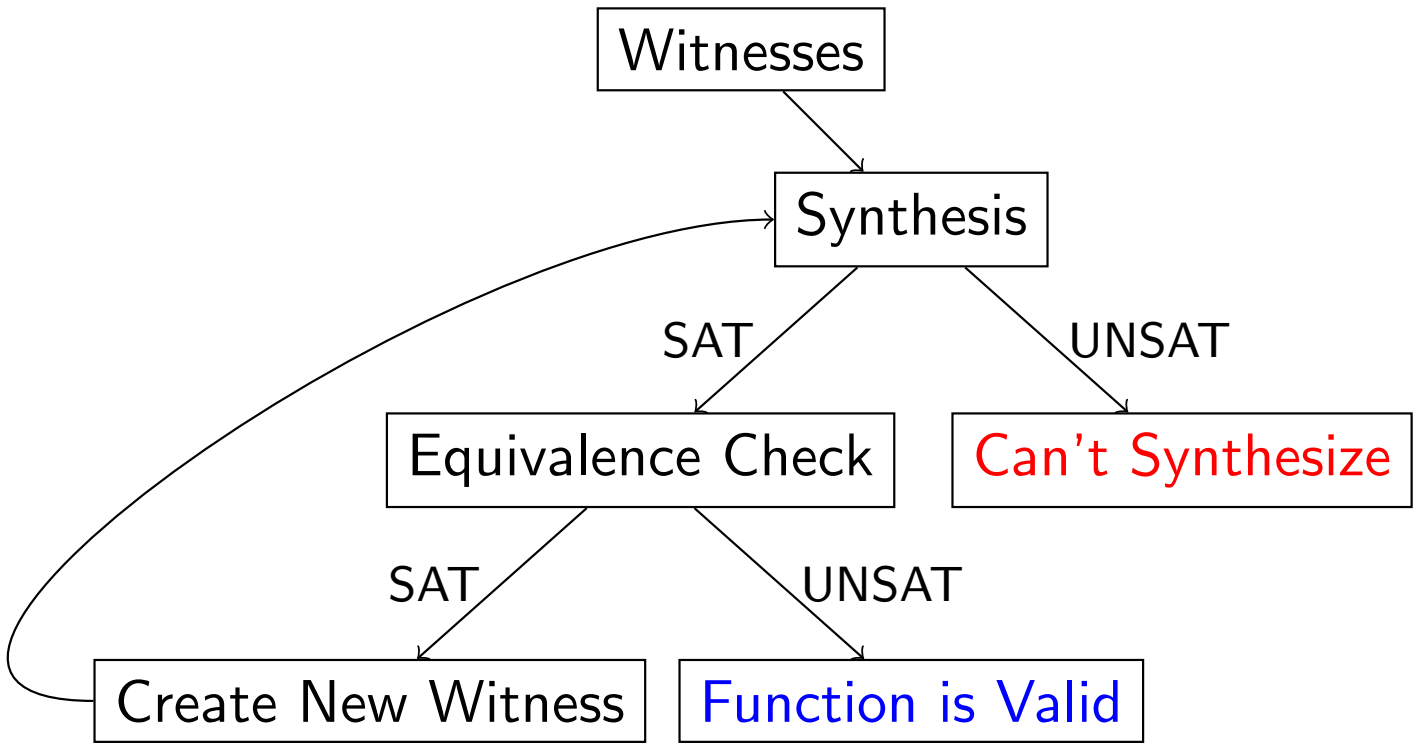
- If the formula is satisfiable, the output is a **counterexample**: a state causing a difference in execution.

al	0x00	vBl	0x88

Metamorphic Extraction

Refinement

- If the function did not match the X86, use the output state to create a new witness. Repeat until error or success.



Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`			

Counter-Example				

- Begin with a program synthesized from the witnesses.
- Try to find a counter-example.

Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`			

Counter-Example	`vBl = 0x88` `al = 0x00`			

- Synthesize a program from the witnesses.
- Try to find a counter-example.
  - Found: generate new witness.

Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`
		`a = al + vBl;`
		`b = a ^ 0xD9;`
		`c = b - 0xE8;`
		`d = vBl + c;`		

Counter-Example	`vBl = 0x88` `al = 0x00`			

- Synthesize a program from the witnesses.
- Try to find a counter-example.

Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`
		`a = al + vBl;`
		`b = a ^ 0xD9;`
		`c = b - 0xE8;`
		`d = vBl + c;`		

Counter-Example	`vBl = 0x88` `al = 0x00`	`vBl = 0x20` `al = 0x10`		

- Synthesize a program from the witnesses.
- Try to find a counter-example.
  - Found: generate new witness.

Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`
		`a = al + vBl;`
		`b = a ^ 0xD9;`
		`c = b - 0xE8;`
		`d = vBl + c;`
			`a = al + vBl;`
			`b = a ^ 0xD3;`
			`c = b + 0x0E;`
			`d = vBl + c;`	

Counter-Example	`vBl = 0x88` `al = 0x00`	`vBl = 0x20` `al = 0x10`		

- Synthesize a program from the witnesses.
- Try to find a counter-example.

Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`
		`a = al + vBl;`
		`b = a ^ 0xD9;`
		`c = b - 0xE8;`
		`d = vBl + c;`
			`a = al + vBl;`
			`b = a ^ 0xD3;`
			`c = b + 0x0E;`
			`d = vBl + c;`

Counter-Example	`vBl = 0x88` `al = 0x00`	`vBl = 0x20` `al = 0x10`	`vBl = 0x23` `al = 0x08`

- Synthesize a program from the witnesses.
- Try to find a counter-example.
  - Found: generate new witness.

Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`
		`a = al + vBl;`
		`b = a ^ 0xD9;`
		`c = b - 0xE8;`
		`d = vBl + c;`
			`a = al + vBl;`
			`b = a ^ 0xD3;`
			`c = b + 0x0E;`
			`d = vBl + c;`
				`a = al + vBl;`
				`b = a ^ 0xD2;`
				`c = b + 0x0F;`
				`d = vBl + c;`

Counter-Example	`vBl = 0x88` `al = 0x00`	`vBl = 0x20` `al = 0x10`	`vBl = 0x23` `al = 0x08`	

- Synthesize a program from the witnesses.
- Try to find a counter-example.

Metamorphic Extraction

In Action

```
lodsb
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

The deobfuscated sequence is shown for clarity. Our analysis works upon obfuscated sequences.

Synthesized Program	`a = al ^ vBl;`
	`b = a ^ 0x4A;`
	`c = b ^ 0x85;`
	`d = vBl + c;`
		`a = al + vBl;`
		`b = a ^ 0xD9;`
		`c = b - 0xE8;`
		`d = vBl + c;`
			`a = al + vBl;`
			`b = a ^ 0xD3;`
			`c = b + 0x0E;`
			`d = vBl + c;`
				`a = al + vBl;`
				`b = a ^ 0xD2;`
				`c = b + 0x0F;`
				`d = vBl + c;`

Counter-Example	`vBl = 0x88` `al = 0x00`	`vBl = 0x20` `al = 0x10`	`vBl = 0x23` `al = 0x08`	NONE

- Synthesize a program from the witnesses.
- Try to find a counter-example.
  - Not found: function is valid.

New Course Offering

New training course offering on SMT-based binary program analysis.

- Written for low-level people comfortable programming in Python; no particular math or CS background required.
- Learn what SMT solvers are and how to use them.
- Lecture material vividly illustrated like these slides.
- Students construct a minimal, yet fully functional SMT-based program analysis framework in Python.
  - Dozens of small, guided programming exercises.
  - Code an SMT solver, X86 $\mapsto$ IR translator, ROP compiler[1].
- Available now!

[1] ROP compiler application subject to potential replacement pending forthcoming regulation of the computer security industry

Questions?

rolf@msreverseengineering.com

Check out Möbius Strip Reverse Engineering at:

http://www.msreverseengineering.com

- Program analysis training classes
- Reverse engineering training classes
- Consulting services
- Blog, research archive, and other resources

Thanks

My proofreaders are awesome:

- Igor Skochinsky
- William Whistler
- Vijay D'Silva

People who publish inspiring work are awesome, too.

References


Sorav Bansal and Alex Aiken.
Automatic generation of peephole superoptimizers.
In *ACM Sigplan Notices*, volume 41, pages 394–403. ACM, 2006.


Patrice Godefroid and Ankur Taly.
Automated synthesis of symbolic instruction encodings from i/o samples.
In *ACM SIGPLAN Notices*, volume 47, pages 441–452. ACM, 2012.


Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan.
Synthesis of loop-free programs.
In *ACM SIGPLAN Notices*, volume 46, pages 62–73. ACM, 2011.