

# Program Synthesis in Reverse Engineering

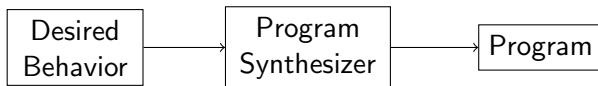
Rolf Rolles  
Möbius Strip Reverse Engineering

<http://www.msreverseengineering.com>

No Such Conference Keynote Speech  
November 19th, 2014

# Program Synthesis in Reverse Engineering

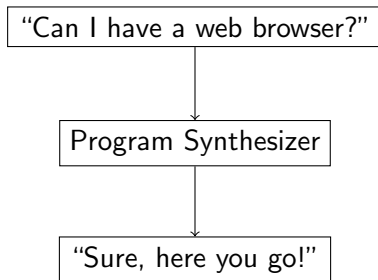
## Overview



- ▶ **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.

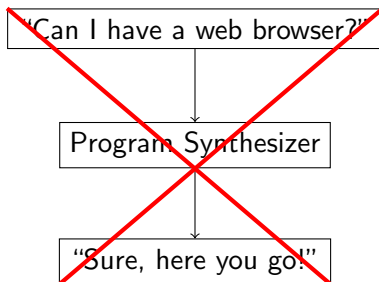
# Program Synthesis

## 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:

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

# Program Synthesis

## The Simplest Possible Program Synthesizer

- ▶ Starting from a behavioral specification:

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

- ▶ 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:

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

- ▶ 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:

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

- ▶ 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:

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

- ▶ 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:

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

- ▶ 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:

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

- ▶ 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:

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

- ▶ 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.

## Introduction

### Ingredients

X86 Disassembly and Assembly

IR Translation

IR Interpreter

SMT Integration

### Applications

## Enumerative Program Synthesis

CPU Emulator Synthesis

Peephole Superdeobfuscation

## Template-Based Program Synthesis

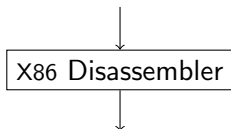
Metamorphic Extraction

## Conclusion

# Ingredients

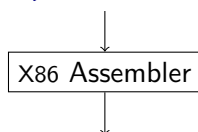
## X86 Disassembly and Assembly

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



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

```
sub bl, bl
movzx ebx, bl
add ebx, 0BBBBBBBh
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

add eax, ecx



IR Translator



```
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

IR for add eax, ecx

```
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;
```

Input State

eax	645EDE7Bh	zf	0	of	0	sf	1
ecx	0FCD02BDEh	af	0	pf	1	cf	0

IR Interpreter

eax	612F0A59h	zf	0	of	0	sf	0
ecx	0FCD02BDEh	af	1	pf	1	cf	1

Output State

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

# Ingredients

## SMT Integration

IR for add eax, ecx

```
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;
```

Query

```
vZF == 1
```

SMT Solver

```
vEax = 0, vEcx = 0
```

- ▶ **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`.

## Introduction

Ingredients

Applications

## Enumerative Program Synthesis

CPU Emulator Synthesis

Peephole Superdeobfuscation

## Template-Based Program Synthesis

Metamorphic Extraction

## Conclusion

# Program Synthesis in Reverse Engineering

## CPU Emulator Synthesis

add eax, ecx

X86 Assembler

Templates

CPU  
Emulator  
Synthesizer

## 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

### Obfuscated Code

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

Deobfuscator  
Generator

### Deobfuscator Rule

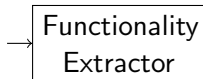
```
push r321
mov r321, i321
mov r322, i322
add r322, r321
pop r321
-----
Simplifies To
-----
mov r322, i321+i322
```

- ▶ 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
```



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

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

## Introduction

### Ingredients

X86 Disassembly and Assembly

IR Translation

IR Interpreter

SMT Integration

### Applications

## Enumerative Program Synthesis

CPU Emulator Synthesis

Peephole Superdeobfuscation

## Template-Based Program Synthesis

Metamorphic Extraction

## Conclusion

# 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.

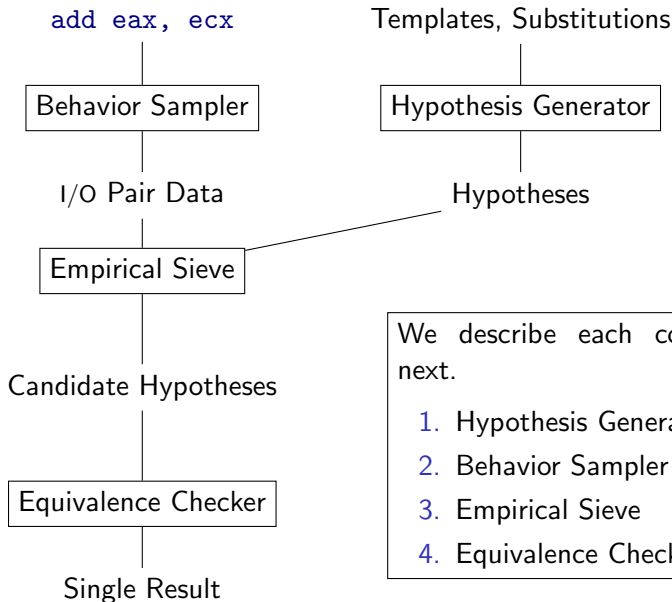
add eax, ecx

↓  
CPU Emulator Synthesizer  
↓

```
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



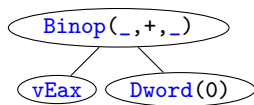
We describe each component next.

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
<code>+</code>	<code>+, -, \&amp;,  , ^</code>
<code>vEax</code>	<code>vEax, vEcx</code>

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

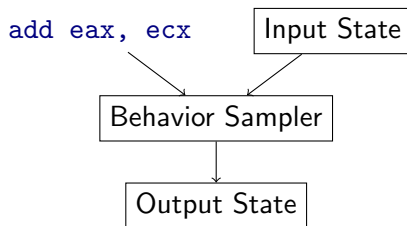
- ▶ A **result location** is a location modified by an instruction.
- ▶ A **hypothesis** is an IR expression of equality between a result location variable and a generated expression.
- ▶ For the instruction `add eax, ebx`, the result locations are  $vEax_{out}$ ,  $vZF_{out}$ ,  $vSF_{out}$ ,  $vPF_{out}$ ,  $vCF_{out}$ ,  $vOF_{out}$ , and  $vAF_{out}$ .

$$\begin{array}{l|l} vZF_{out} == (vEax + vEbx) <s 0 & vZF_{out} == ((vEax + vEbx) \wedge vEax) >u 0 \\ vSF_{out} == (vEax + vEbx) <s 0 & vSF_{out} == ((vEax + vEbx) \wedge vEax) >u 0 \\ vPF_{out} == (vEax + vEbx) <s 0 & vPF_{out} == ((vEax + vEbx) \wedge vEax) >u 0 \\ vCF_{out} == (vEax + vEbx) <s 0 & vCF_{out} == ((vEax + vEbx) \wedge vEax) >u 0 \\ vOF_{out} == (vEax + vEbx) <s 0 & vOF_{out} == ((vEax + vEbx) \wedge vEax) >u 0 \\ vAF_{out} == (vEax + vEbx) <s 0 & vAF_{out} == ((vEax + vEbx) \wedge vEax) >u 0 \end{array}$$

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

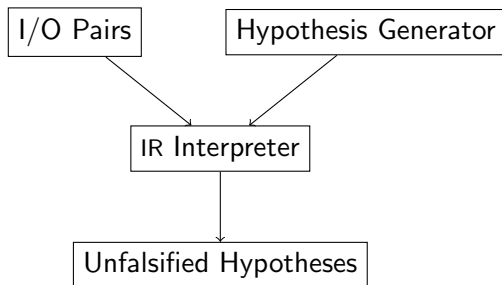
## Empirical Sieve

$eax_{in} = 0$		$0 == (0 == 0)$
$ecx_{in} = 0$	$CF_{out} == (vEax == vEcx)$	$0 == 1$
$cf_{out} = 0$	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,  $CF_{out} == (vEax == vEcx)$  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.

$$\begin{array}{ll} (\mathbf{vEax+vEcx}) == 0 & (\mathbf{vEcx+vEax+vEcx+vEax}) == 0 \\ \text{Expression \#1} & \text{Expression \#2} \end{array}$$

Two hypotheses for  $ZF_{out}$ .

- ▶ We query an SMT solver as to the satisfiability of  $((\mathbf{vEax+vEcx}) == 0) \neq ((\mathbf{vEcx+vEax+vEcx+vEax}) == 0)$ .
  - ▶ If UNSAT, the hypotheses always behave the same.
  - ▶ IF SAT, the solution contains values for  $\mathbf{vEax}$  and  $\mathbf{vEcx}$  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  $zf_{out}$  evaluates to 0, and the other to 1.

$$eax_{in} = 3d800000h$$

$$ecx_{in} = 42800000h$$

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  $zf_{out} = 1$ , then  $(vEax+vEcx) == 0$  is false.
  - ▶ If  $zf_{out} = 0$ , then  $(vEcx+vEax+vEcx+vEax) == 0$  is false.
    - ▶ Hypothesis #2 is falsified.

# CPU Emulator Synthesis

## Behavior Sampler

$\langle out \rangle == \langle in \rangle,$   
 $\langle out \rangle == \langle in \rangle \langle + \rangle \langle in \rangle,$

...

Hypothesis Generator

$CF_{out} == (vEax == vEcx),$   
 $CF_{out} == (vEcx <s 0),$   
 $CF_{out} == (vEax + vEcx <u vEax),$

...

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

# CPU Emulator Synthesis

## Hypothesis Generator

add eax, ecx

Behavior Sampler

I/O Pair Data

$\langle out \rangle == \langle in \rangle,$   
 $\langle out \rangle == \langle in \rangle \langle + \rangle \langle in \rangle,$

...

Hypothesis Generator

$CF_{out} == (vEax == vEcx),$   
 $CF_{out} == (vEcx <_s 0),$   
 $CF_{out} == (vEax + vEcx <_u vEax),$

...

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

# CPU Emulator Synthesis

## Empirical Sieve

add eax, ecx

Behavior Sampler

I/O Pair Data

Empirical Sieve

$CF_{out} == (vEcx <s 0),$   
 $CF_{out} == (vEax + vEcx <u vEax),$   
...

$\langle out \rangle == \langle in \rangle,$   
 $\langle out \rangle == \langle in \rangle <+ \rangle \langle in \rangle,$

...

Hypothesis Generator

$CF_{out} == (vEax == vEcx),$   
 $CF_{out} == (vEcx <s 0),$   
 $CF_{out} == (vEax + vEcx <u vEax),$   
...

Remove hypotheses  
that are shown to be  
false by any I/O pair.

# CPU Emulator Synthesis

## Equivalence Checking

add eax, ecx

Behavior Sampler

I/O Pair Data

Empirical Sieve

$CF_{out} == (vEcx <_s 0),$   
 $CF_{out} == (vEax + vEcx <_u vEax),$

...

Equivalence Checker

$CF_{out} == vEax + vEcx <_u vEax$

$\langle out \rangle == \langle in \rangle,$   
 $\langle out \rangle == \langle in \rangle \langle + \rangle \langle in \rangle,$

...

Hypothesis Generator

$CF_{out} == (vEax == vEcx),$   
 $CF_{out} == (vEcx <_s 0),$   
 $CF_{out} == (vEax + vEcx <_u vEax),$

...

The equivalence checker removes all but one hypothesis.

## Introduction

### Ingredients

X86 Disassembly and Assembly

IR Translation

IR Interpreter

SMT Integration

### Applications

## Enumerative Program Synthesis

CPU Emulator Synthesis

Peephole Superdeobfuscation

## Template-Based Program Synthesis

Metamorphic Extraction

## Conclusion

# Peephole Superdeobfuscation

## Rule-Based Obfuscation

### Obfuscator Rules

Unobfuscated	Obfuscated	Constraints
<code>add r8<sub>1</sub>, r8<sub>2</sub></code>	<code>add r8<sub>1</sub>, imm8</code> <code>add r8<sub>1</sub>, r8<sub>2</sub></code> <code>sub r8<sub>1</sub>, imm8</code>	
<code>sub r32<sub>1</sub>, r32<sub>2</sub></code>	<code>push r32<sub>3</sub></code> <code>mov r32<sub>3</sub>, r32<sub>2</sub></code> <code>sub r32<sub>1</sub>, r32<sub>3</sub></code> <code>pop r32<sub>3</sub></code>	<code>r32<sub>3</sub> ≠ r32<sub>1</sub></code> <code>r32<sub>1</sub> ≠ esp</code> <code>r32<sub>1</sub> ≠ esp</code> <code>r32<sub>3</sub> ≠ esp</code>

- ▶ 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.

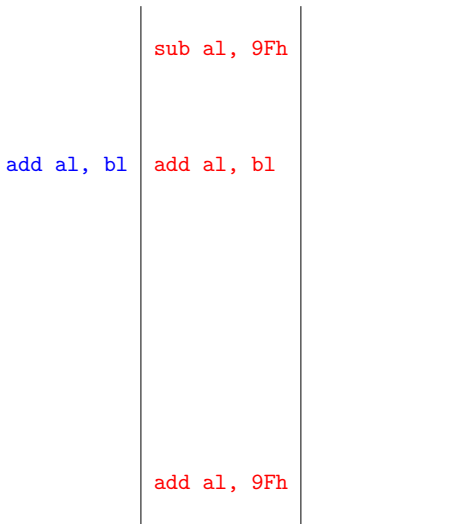
# Peephole Superdeobfuscation

```
add a1, b1
```



# Peephole Superdeobfuscation

Obfuscation →



Replace **instructions** with **obfuscated sequences**

# Peephole Superdeobfuscation

Obfuscation →

<code>sub al, 9Fh</code>	<code>sub al, 9Fh</code>
	<code>sub al, 7Fh</code>
<code>add al, bl</code>	<code>add al, bl</code>
	<code>add al, 7Fh</code>
	<code>push edx</code>
	<code>mov dh, 22h</code>
	<code>inc dh</code>
	<code>dec dh</code>
	<code>add dh, 7Dh</code>
<code>add al, 9Fh</code>	<code>add al, dh</code>
	<code>pop edx</code>

Replace `instructions` with `obfuscated sequences`

# Peephole Superdeobfuscation

Obfuscation →

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

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

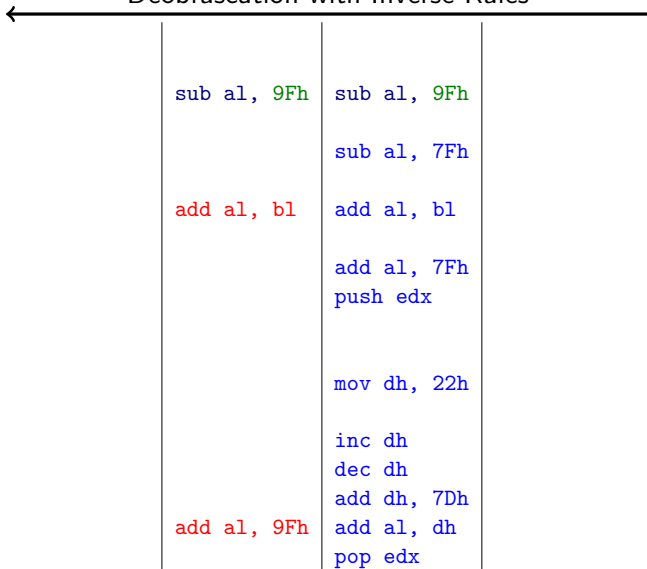
## Deobfuscation with Inverse Rules

			push cx
			mov ch, 9Fh
		sub al, 9Fh	sub al, ch
			pop cx
		sub al, 7Fh	sub al, 7Fh
			sub al, 70h
		add al, b1	add al, b1
			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

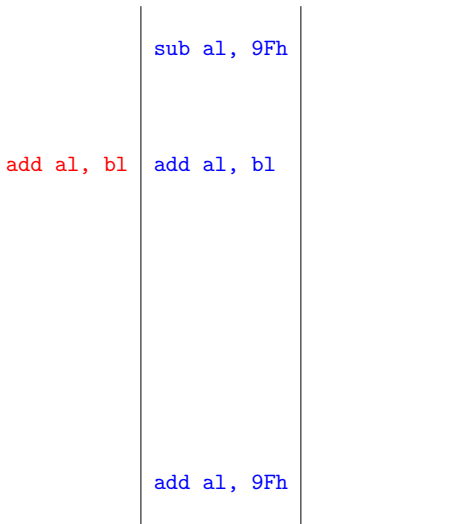
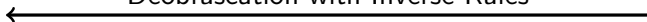
## Deobfuscation with Inverse Rules



Replace **obfuscated** sequences with **instructions**

# Peephole Superdeobfuscation

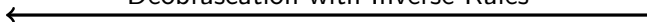
Deobfuscation with Inverse Rules



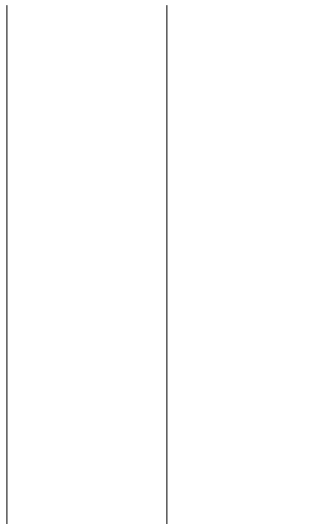
Replace **obfuscated sequences** with **instructions**

# Peephole Superdeobfuscation

Deobfuscation with Inverse Rules



add a1, b1





# Peephole Superdeobfuscation

## Rule-Based Deobfuscation

### Deobfuscator Rules

Obfuscated	Unobfuscated	Constraints
<code>add r8<sub>1</sub>, imm8</code> <code>add r8<sub>1</sub>, r8<sub>2</sub></code> <code>sub r8<sub>1</sub>, imm8</code>	<code>add r8<sub>1</sub>, r8<sub>2</sub></code>	
<code>push r32<sub>3</sub></code> <code>mov r32<sub>3</sub>, r32<sub>2</sub></code> <code>sub r32<sub>1</sub>, r32<sub>3</sub></code> <code>pop r32<sub>3</sub></code>	<code>sub r32<sub>1</sub>, r32<sub>2</sub></code>	$r32_3 \neq r32_1$ $r32_1 \neq \text{esp}$ $r32_1 \neq \text{esp}$ $r32_3 \neq \text{esp}$

- ▶ 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

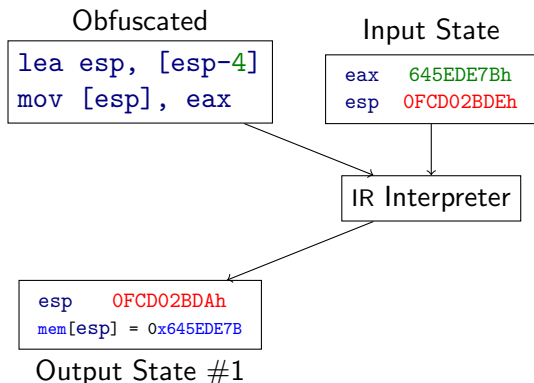
Obfuscated

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

Collect obfuscated sequences.

# Peephole Superdeobfuscation

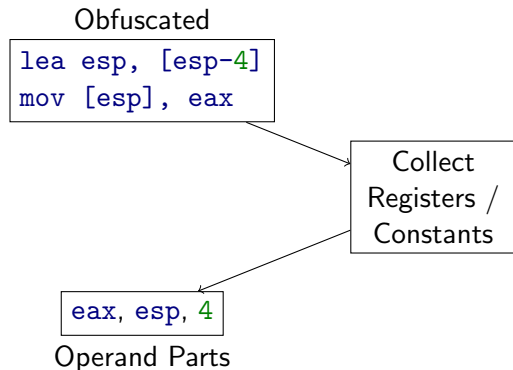
The Idea: Obfuscated/Unobfuscated Behavior Must Match



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

## Candidate Templates

```
push r32
-----
push i32
-----
add r32, i32
```

```
eax, esp, 4
```

Operand Parts

Candidate  
Enumerator

Candidate

```
push eax
```

```
push esp
```

```
push 4
```

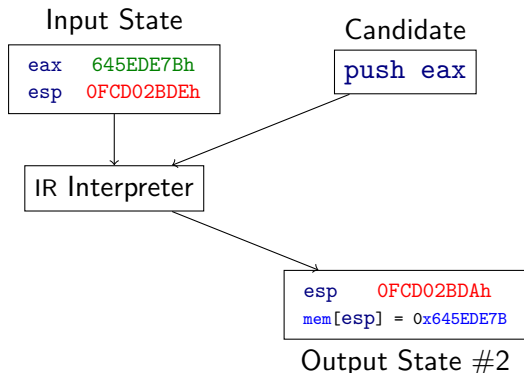
```
add esp, 4
```

```
add eax, 4
```

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

# Peephole Superdeobfuscation

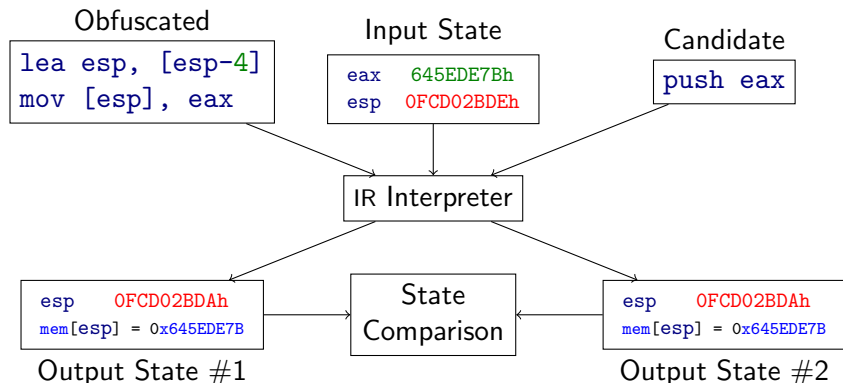
The Idea: Obfuscated/Unobfuscated Behavior Must Match



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

# Peephole Superdeobfuscation

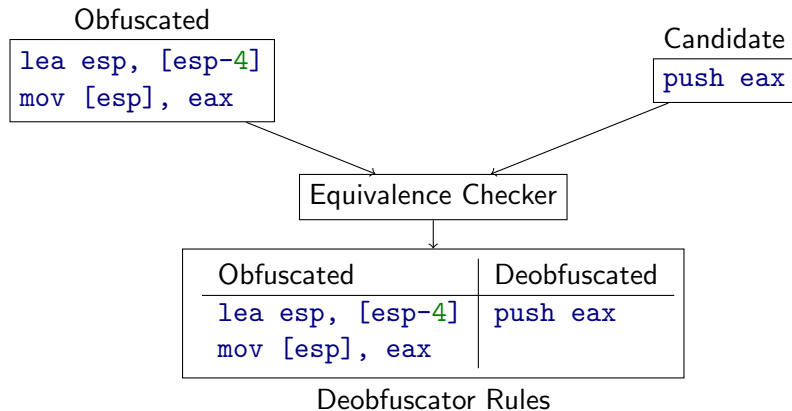
The Idea: Obfuscated/Unobfuscated Behavior Must Match



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
<code>lea esp, [esp-4]</code> <code>mov [esp], eax</code>	<code>push eax</code>	

- ▶ 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
<code>lea esp, [esp-4]</code> <code>mov [esp], eax</code>	<code>push eax</code>	

### Generalized Rules

<code>lea esp, [esp-4]</code> <code>mov [esp], r32</code>	<code>push r32</code>	<code>r32 ≠ esp</code>
--	-----------------------	------------------------

- ▶ 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
<pre>▶ sub esp, 4 ▶ mov [esp], r32</pre>	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
<code>sub esp, 4</code> <code>mov [esp], r32</code>	<code>push r32</code>	$r32 \neq esp$

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
<pre>sub esp, 4 mov [esp], r32</pre>	<pre>push r32</pre>	$r32 \neq esp$

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
<pre>sub esp, 4 mov [esp], r32</pre>	<pre>push r32</pre>	$r32 \neq esp$

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	$r32 \neq esp$
▶ push r32 <sub>1</sub> ▶ mov [esp], r32 <sub>2</sub>	push r32 <sub>2</sub>	$r32_2 \neq esp$

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	$r32 \neq esp$
push r32 <sub>1</sub> mov [esp], r32 <sub>2</sub>	push r32 <sub>2</sub>	$r32_2 \neq esp$

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 <sub>1</sub> mov [esp], r32 <sub>2</sub>	push r32 <sub>2</sub>	$r32_2 \neq esp$
▶	push r32 <sub>1</sub> ▶ xor [esp], i32 ▶ pop r32 <sub>2</sub>	mov r32 <sub>2</sub> , r32 <sub>1</sub> xor r32 <sub>2</sub> , i32	$r32_1 \neq esp$ $r32_2 \neq esp$
▶	xor [esp], 7FEE03B1h		
▶	pop ebp		
	sub esp, 4		
	mov [esp], esi		
	push edx		
	mov edx, 6F6B5081h		
	mov esi, 0BC6D38Bh		
	add esi, edx		
	pop edx		

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
	<pre>sub esp, 4 mov [esp], r32</pre>	<pre>push r32</pre>	$r32 \neq esp$
	<pre>push r32<sub>1</sub> mov [esp], r32<sub>2</sub></pre>	<pre>push r32<sub>2</sub></pre>	$r32_2 \neq esp$
	<pre>push r32<sub>1</sub> xor [esp], i32 pop r32<sub>2</sub></pre>	<pre>mov r32<sub>2</sub>, r32<sub>1</sub> xor r32<sub>2</sub>, i32</pre>	$r32_1 \neq esp$ $r32_2 \neq esp$

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
	<pre>▶ sub esp, 4 ▶ mov [esp], r32</pre>	<pre>push r32</pre>	$r32 \neq esp$
	<pre>push r32<sub>1</sub> mov [esp], r32<sub>2</sub></pre>	<pre>push r32<sub>2</sub></pre>	$r32_2 \neq esp$
	<pre>push r32<sub>1</sub> xor [esp], i32 pop r32<sub>2</sub></pre>	<pre>mov r32<sub>2</sub>, r32<sub>1</sub> xor r32<sub>2</sub>, i32</pre>	$r32_1 \neq esp$ $r32_2 \neq esp$

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
push eax	sub esp, 4 mov [esp], r32	push r32	$r32 \neq esp$
mov eax, ebp	push r32 <sub>1</sub> mov [esp], r32 <sub>2</sub>	push r32 <sub>2</sub>	$r32_2 \neq esp$
push ebp	push r32 <sub>1</sub> xor [esp], i32 pop r32 <sub>2</sub>	mov r32 <sub>2</sub> , r32 <sub>1</sub> xor r32 <sub>2</sub> , i32	$r32_1 \neq esp$ $r32_2 \neq esp$
mov ebp, eax			
xor ebp, 7FEE03B1h			
▶ push esi			

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
	<pre>sub esp, 4 mov [esp], r32</pre>	<pre>push r32</pre>	$r32 \neq esp$
	<pre>push r32<sub>1</sub> mov [esp], r32<sub>2</sub></pre>	<pre>push r32<sub>2</sub></pre>	$r32_2 \neq esp$
	<pre>push r32<sub>1</sub> xor [esp], i32 pop r32<sub>2</sub></pre>	<pre>mov r32<sub>2</sub>, r32<sub>1</sub> xor r32<sub>2</sub>, i32</pre>	$r32_1 \neq esp$ $r32_2 \neq esp$

No simplification.

# Peephole Superdeobfuscation

In Action

	Obfuscated	Deobfuscated	Constraints
<code>push eax</code>	<code>sub esp, 4 mov [esp], r32</code>	<code>push r32</code>	$r32 \neq esp$
<code>mov eax, ebp</code>	<code>push r32<sub>1</sub> mov [esp], r32<sub>2</sub></code>	<code>push r32<sub>2</sub></code>	$r32_2 \neq esp$
<code>push ebp</code>	<code>push r32<sub>1</sub> xor [esp], i32 pop r32<sub>2</sub></code>	<code>mov r32<sub>2</sub>, r32<sub>1</sub> xor r32<sub>2</sub>, i32</code>	$r32_1 \neq esp$ $r32_2 \neq esp$
<code>mov ebp, eax</code> <code>xor ebp, 7FEE03B1h</code> <code>push esi</code>			

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

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
	<pre>sub esp, 4 mov [esp], r32</pre>	<pre>push r32</pre>	$r32 \neq esp$
	<pre>push r32<sub>1</sub> mov [esp], r32<sub>2</sub></pre>	<pre>push r32<sub>2</sub></pre>	$r32_2 \neq esp$
	<pre>push r32<sub>1</sub> xor [esp], i32 pop r32<sub>2</sub></pre>	<pre>mov r32<sub>2</sub>, r32<sub>1</sub> xor r32<sub>2</sub>, i32</pre>	$r32_1 \neq esp$ $r32_2 \neq esp$

No simplification.



# Peephole Superdeobfuscation

In Action

	Obfuscated	Deobfuscated	Constraints
push eax	sub esp, 4 mov [esp], r32	push r32	$r32 \neq esp$
mov eax, ebp push ebp	push r32 <sub>1</sub> mov [esp], r32 <sub>2</sub>	push r32 <sub>2</sub>	$r32_2 \neq esp$
mov ebp, eax xor ebp, 7FEE03B1h push esi	push r32 <sub>1</sub> xor [esp], i32 pop r32 <sub>2</sub>	mov r32 <sub>2</sub> , r32 <sub>1</sub> xor r32 <sub>2</sub> , i32	$r32_1 \neq esp$ $r32_2 \neq esp$
	▶ push r32 <sub>1</sub> ▶ mov r32 <sub>1</sub> , i32 <sub>1</sub> ▶ mov r32 <sub>2</sub> , i32 <sub>2</sub> ▶ add r32 <sub>2</sub> , r32 <sub>1</sub> ▶ pop r32 <sub>1</sub>	mov r32 <sub>2</sub> , i32 <sub>1</sub> +i32 <sub>2</sub>	$r32_1 \neq esp$ $r32_2 \neq esp$ $r32_1 \neq r32_2$

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

Learn rule and generalize.

# Peephole Superdeobfuscation

In Action

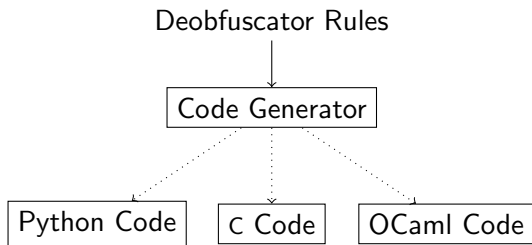
	Obfuscated	Deobfuscated	Constraints
push eax	sub esp, 4 mov [esp], r32	push r32	$r32 \neq esp$
mov eax, ebp push ebp	push r32 <sub>1</sub> mov [esp], r32 <sub>2</sub>	push r32 <sub>2</sub>	$r32_2 \neq esp$
	push r32 <sub>1</sub> xor [esp], i32 pop r32 <sub>2</sub>	mov r32 <sub>2</sub> , r32 <sub>1</sub> xor r32 <sub>2</sub> , i32	$r32_1 \neq esp$ $r32_2 \neq esp$
mov ebp, eax xor ebp, 7FEE03B1h push esi	push r32 <sub>1</sub> mov r32 <sub>1</sub> , i32 <sub>1</sub> mov r32 <sub>2</sub> , i32 <sub>2</sub> add r32 <sub>2</sub> , r32 <sub>1</sub> pop r32 <sub>1</sub>	mov r32 <sub>2</sub> , i32 <sub>1</sub> +i32 <sub>2</sub>	$r32_1 \neq esp$ $r32_2 \neq esp$ $r32_1 \neq r32_2$

▶ mov esi, 7B32240Ch

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

```
mov eax, 12h ◀  
or edx, eax ◀  
add eax, 34h ◀  
jmp @end
```

```
bswap ax  
bsr edx, 12h  
salc  
cmc
```

```
@end:  
xor edx, edx ◀  
mov eax, [esp]  
add esp, 4  
ret
```

- ▶ 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.

## Introduction

### Ingredients

X86 Disassembly and Assembly

IR Translation

IR Interpreter

SMT Integration

### Applications

## Enumerative Program Synthesis

CPU Emulator Synthesis

Peephole Superdeobfuscation

## Template-Based Program Synthesis

Metamorphic Extraction

## Conclusion

# Template-Based Program Synthesis

## Overview

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

$y = \sim x;$	$y = -x;$
$z = \sim y;$	$z = \sim y;$
<hr/>	
$y = \sim x;$	$y = -x;$
$z = -y;$	$z = -y;$

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 <code>bop1</code> , <code>bop2</code>	$\exists \text{ bop1, bop2} \in \mathbf{Bool} \cdot$
Such that, for all values of <code>x</code>	$\forall \text{ x} \in \mathbf{BV}[32] \cdot$
In the code	
<code>y = bop1 ? -x : ~x</code>	let <code>y = bop1 ? -x : ~x</code> in
<code>z = bop2 ? -y : ~y</code>	let <code>z = bop2 ? -y : ~y</code> in
<code>z == x+1</code> is always true?	<code>z == x+1</code>

- ▶ 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;  
}
```

## Introduction

### Ingredients

X86 Disassembly and Assembly

IR Translation

IR Interpreter

SMT Integration

### Applications

## Enumerative Program Synthesis

CPU Emulator Synthesis

Peephole Superdeobfuscation

## Template-Based Program Synthesis

Metamorphic Extraction

## Conclusion

# 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

```
lods b
add al, bl
xor al, 0D2h
sub al, 0F1h
add bl, al
```

→ Obfuscator →

```
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

```
a1 = Load(vMem, vEdi, 8);
a = op1 == 0 ? a1 + vB1 : op1 == 1 ? a1 - vB1 : a1 ^ vB1;
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 ? vB1 + c : op4 == 1 ? vB1 - c : vB1 ^ c;
```

#### ▶ X86-related variables:

- ▶ `a1` is the value read by `lodsrb`
- ▶ `vB1` is the initial value of `bl`
- ▶ `d` is the final value of `bl`

#### ▶ Template parameters:

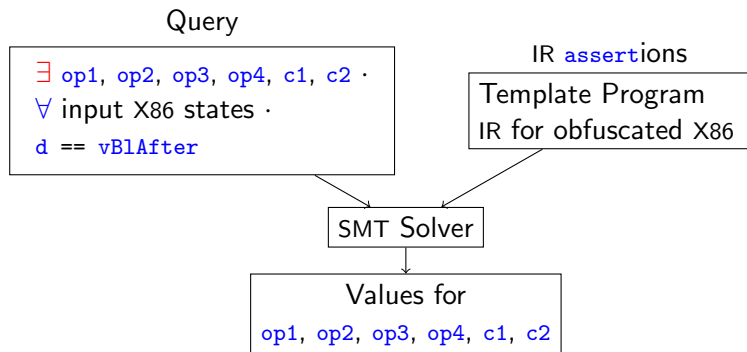
- ▶ Constants `c1`, `c2`
- ▶ Operators `op1`, `op2`, `op3`, `op4`

▶ 0:add, 1:sub, 2+:xor

```
lodsrb
op al, bl
op al, i81
op al, i82
op bl, al
```

# Metamorphic Extraction

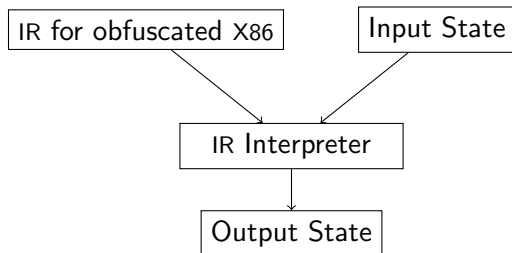
Straightforward Approach Using  $\exists \cdot \forall$ .



- ▶ 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 sequence.
- ▶ 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

a1	0x00	◀	vB1	0x00	◀	vB1After	0xE1	◀
----	------	---	-----	------	---	----------	------	---

Into the template

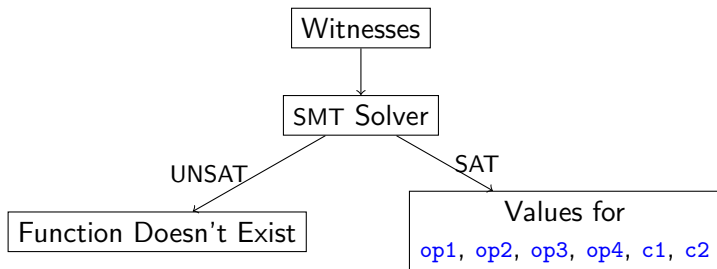
```
a = op1 == 0 ? a1 + vB1 : op1 == 1 ? a1 - vB1 : a1 ^ vB1; ▶ ◀  
b = op2 == 0 ? a + c1 : op2 == 1 ? a - c1 : a ^ c1;  
c = op3 == 0 ? b + c2 : op3 == 1 ? b - c2 : b ^ c2;  
d = op4 == 0 ? vB1 + c : op4 == 1 ? vB1 - c : vB1 ^ c; ◀
```

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;
```

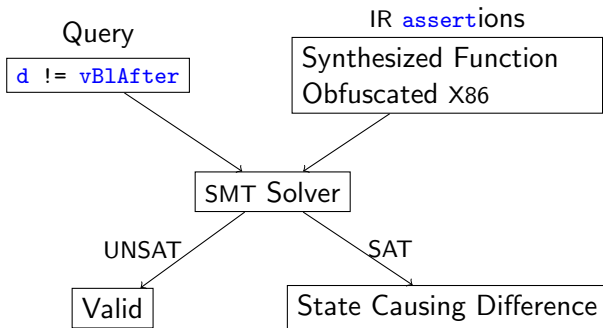
```
c = b ^ 0x85;
```

```
d = vB1 + c;
```

# Metamorphic Extraction

## Equivalence Checking

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



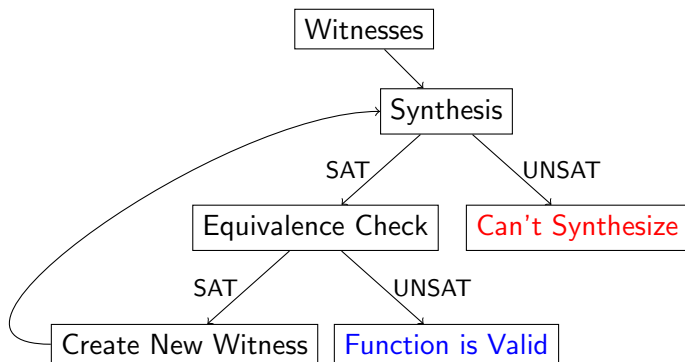
- ▶ If the formula is satisfiable, the output is a **counterexample**: a state causing a difference in execution.

a1	0x00	vB1	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

```
lods b
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	<pre>a = al ^ vBl; b = a ^ 0x4A; c = b ^ 0x85; d = vBl + c;</pre>			
Counter-Example				

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

# Metamorphic Extraction

## In Action

```
lods b
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	$a = al \oplus vBl;$ $b = a \oplus 0x4A;$ $c = b \oplus 0x85;$ $d = vBl + c;$			
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

```
lods b
add a1, b1
xor a1, 0D2h
sub a1, 0F1h
add b1, a1
```

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

Synthesized Program	$a = a1 \oplus vB1;$	$a = a1 + vB1;$		
	$b = a \oplus 0x4A;$	$b = a \oplus 0xD9;$		
	$c = b \oplus 0x85;$	$c = b - 0xE8;$		
	$d = vB1 + c;$	$d = vB1 + c;$		
Counter-Example	$vB1 = 0x88$			
	$a1 = 0x00$			

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

# Metamorphic Extraction

## In Action

```
lods b
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	$a = al \oplus vBl;$	$a = al + vBl;$	
	$b = a \oplus 0x4A;$	$b = a \oplus 0xD9;$	
	$c = b \oplus 0x85;$	$c = b - 0xE8;$	
	$d = vBl + c;$	$d = vBl + c;$	
Counter-Example	$vBl = 0x88$	$vBl = 0x20$	
	$al = 0x00$	$al = 0x10$	

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



# Metamorphic Extraction

## In Action

```
lods b
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	$a = al \oplus vBl;$ $b = a \oplus 0x4A;$ $c = b \oplus 0x85;$ $d = vBl + c;$	$a = al + vBl;$ $b = a \oplus 0xD9;$ $c = b - 0xE8;$ $d = vBl + c;$	$a = al + vBl;$ $b = a \oplus 0xD3;$ $c = b + 0x0E;$ $d = vBl + c;$
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

```
lods b
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	$a = al \oplus vBl;$ $b = a \oplus 0x4A;$ $c = b \oplus 0x85;$ $d = vBl + c;$	$a = al + vBl;$ $b = a \oplus 0xD9;$ $c = b - 0xE8;$ $d = vBl + c;$	$a = al + vBl;$ $b = a \oplus 0xD3;$ $c = b + 0xE;$ $d = vBl + c;$
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

```
lods b
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	$a = al \oplus vB1;$ $b = a \oplus 0x4A;$ $c = b \oplus 0x85;$ $d = vB1 + c;$	$a = al + vB1;$ $b = a \oplus 0xD9;$ $c = b - 0xE8;$ $d = vB1 + c;$	$a = al + vB1;$ $b = a \oplus 0xD3;$ $c = b + 0x0E;$ $d = vB1 + c;$	$a = al + vB1;$ $b = a \oplus 0xD2;$ $c = b + 0x0F;$ $d = vB1 + c;$
Counter-Example	$vB1 = 0x88$ $al = 0x00$	$vB1 = 0x20$ $al = 0x10$	$vB1 = 0x23$ $al = 0x08$	

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

# Metamorphic Extraction

## In Action

```
lods b
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	$a = al \oplus vB1;$ $b = a \oplus 0x4A;$ $c = b \oplus 0x85;$ $d = vB1 + c;$	$a = al + vB1;$ $b = a \oplus 0xD9;$ $c = b - 0xE8;$ $d = vB1 + c;$	$a = al + vB1;$ $b = a \oplus 0xD3;$ $c = b + 0x0E;$ $d = vB1 + c;$	$a = al + vB1;$ $b = a \oplus 0xD2;$ $c = b + 0x0F;$ $d = vB1 + c;$
Counter-Example	$vB1 = 0x88$ $al = 0x00$	$vB1 = 0x20$ $al = 0x10$	$vB1 = 0x23$ $al = 0x08$	NONE

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

▶ Not found: function is valid.

## Introduction

### Ingredients

X86 Disassembly and Assembly

IR Translation

IR Interpreter

SMT Integration

### Applications

## Enumerative Program Synthesis

CPU Emulator Synthesis

Peephole Superdeobfuscation

## Template-Based Program Synthesis

Metamorphic Extraction

## Conclusion

# 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<sup>1</sup>.
- ▶ Available now!

---

<sup>1</sup>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.