Synesthesia: A Modern Approach to Shellcode Generation

Rolf Rolles
Möbius Strip Reverse Engineering

http://www.msreverseengineering.com

November 8th, 2016
Overview

Symbolic Program Synthesis

Extensions

Discussion

Conclusion
Synesthesia

Background

- This idea percolated in my mind for four years.
  - I was too busy to try it out.
- Meanwhile, YICES implemented a specialized solver for the types of equations I needed.
  - Z3 and CVC4 also have suitable, but less specialized, solvers.
- This talk summarizes the results of my experiments.
  - **Mathematically**, the problem is more-or-less solved.
  - **Practically**, there are important limitations at present:
    1. Scalability issues with current solvers.
    2. A remediable deficiency regarding memory accesses.
Synesthesia
Classical Memory-Corruption Exploitation

Program Execution
Synesthesia
Classical Memory-Corruption Exploitation

Malicious Input → Process Accepts Input

Program Execution

Invalid Inputs Discarded
Shellcode Executes
Synesthesia
Classical Memory-Corruption Exploitation

Program Execution
Malicious Input → Process Accepts Input
... Invalid Inputs Discarded
Input Validated → Invalid Inputs Discarded
Synesthesia
Classical Memory-Corruption Exploitation

Program Execution

Malicious Input → Process Accepts Input

Invalid Inputs Discarded

Input Validated

Invalid Inputs Discarded

Input Transformed
Synesthesia
Classical Memory-Corruption Exploitation

Malicious Input → Process Accepts Input

... Input Validated → Invalid Inputs Discarded

... Input Transformed

... Vulnerability Triggered

Program Execution
Synesthesia
Classical Memory-Corruption Exploitation

- Malicious Input
  - Process Accepts Input
    - Input Validated
      - Input Transformed
        - Vulnerability Triggered
          - Input Executed
            - Program Execution
              - Invalid Inputs Discarded
                - Shellcode Executes
## Synesthesia

### Restrictions on the Shellcode

<table>
<thead>
<tr>
<th>Input is restricted by ...</th>
<th>Restriction placed on shellcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passed to <code>strcpy()</code></td>
<td>No NULL bytes allowed</td>
</tr>
<tr>
<td>Passed to <code>strupr()</code></td>
<td>All ASCII letters become uppercase</td>
</tr>
<tr>
<td>Used as a format string</td>
<td>Use of ‘%’ character dicey</td>
</tr>
<tr>
<td>Bytes passed to <code>isprint()</code></td>
<td>Bytes must be printable</td>
</tr>
<tr>
<td>Bytes passed to <code>isalnum()</code></td>
<td>Bytes must be alphanumeric</td>
</tr>
</tbody>
</table>

Restrictions are arbitrary and vary per vulnerability.
Synesthesia

Restrictions on the Shellcode

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>PUSH ES</td>
<td>POP ES</td>
</tr>
<tr>
<td>1</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>PUSH SS</td>
<td>POP SS</td>
</tr>
<tr>
<td>2</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG=ES (Prefix)</td>
<td>DAA</td>
</tr>
<tr>
<td>3</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gv, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG=SS (Prefix)</td>
<td>AAA</td>
</tr>
<tr>
<td>4</td>
<td>eAX</td>
<td>eCX</td>
<td>eDX</td>
<td>eBX</td>
<td>eSP</td>
<td>eBP</td>
<td>eSI</td>
<td>eDI</td>
</tr>
<tr>
<td>5</td>
<td>rAX</td>
<td>rCX</td>
<td>rDX</td>
<td>rBX</td>
<td>rSP</td>
<td>rBP</td>
<td>rSI</td>
<td>rDI</td>
</tr>
<tr>
<td>6</td>
<td>PUSHAD</td>
<td>POPAD</td>
<td>BOUND</td>
<td>ARPL</td>
<td>SEG=FS (Prefix)</td>
<td>SEG=GS (Prefix)</td>
<td>Operand Size (Prefix)</td>
<td>Address Size (Prefix)</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>NO</td>
<td>B/NAE/C</td>
<td>NB/AE/NC</td>
<td>Z/E</td>
<td>NZ/NE</td>
<td>BE/NA</td>
<td>NBE/A</td>
</tr>
<tr>
<td>8</td>
<td>Eb, lb</td>
<td>Ev, Iz</td>
<td>Eb, lb</td>
<td>Ev, lb</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
</tr>
</tbody>
</table>

- ▶ Example: restricted to lower-case alphanumeric bytes.
  - ▶ Can't use any of the red opcodes.
- ▶ Situation is even more dire than this slide indicates.
  - ▶ Not only opcode bytes restricted, but also operand bytes.
Various Ways to Set `eax` to `0h`

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 00 00 00 00</td>
<td><code>mov eax, 0</code></td>
</tr>
<tr>
<td>33 C0</td>
<td><code>xor eax, eax</code></td>
</tr>
<tr>
<td>F8</td>
<td><code>clc</code></td>
</tr>
<tr>
<td>1E C0</td>
<td><code>sbb eax, eax</code></td>
</tr>
<tr>
<td>25 56 34 42 24</td>
<td><code>and eax, 24423456h</code></td>
</tr>
<tr>
<td>25 28 48 21 42</td>
<td><code>and eax, 42214828h</code></td>
</tr>
<tr>
<td>6A 30</td>
<td><code>push 30h</code></td>
</tr>
<tr>
<td>58</td>
<td><code>pop eax</code></td>
</tr>
<tr>
<td>34 30</td>
<td><code>xor al, 30h</code></td>
</tr>
</tbody>
</table>

No NULL bytes
Various Ways to Set `eax` to `0h`

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶</td>
<td>B8 00 00 00 00</td>
<td><code>mov eax, 0</code></td>
</tr>
<tr>
<td>▶</td>
<td>33 C0</td>
<td><code>xor eax, eax</code></td>
</tr>
<tr>
<td>▶</td>
<td>F8</td>
<td><code>clc</code></td>
</tr>
<tr>
<td>▶</td>
<td>1E C0</td>
<td><code>sbb eax, eax</code></td>
</tr>
<tr>
<td>▶</td>
<td>25 56 34 42 24</td>
<td><code>and eax, 24423456h</code></td>
</tr>
<tr>
<td>▶</td>
<td>25 28 48 21 42</td>
<td><code>and eax, 42214828h</code></td>
</tr>
<tr>
<td>▶</td>
<td>6A 30</td>
<td><code>push 30h</code></td>
</tr>
<tr>
<td>▶</td>
<td>58</td>
<td><code>pop eax</code></td>
</tr>
<tr>
<td>▶</td>
<td>34 30</td>
<td><code>xor al, 30h</code></td>
</tr>
</tbody>
</table>

No ‘%’ (25) bytes
## Various Ways to Set `eax` to 0h

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 00 00 00 00</td>
<td><code>mov eax, 0</code></td>
</tr>
<tr>
<td>33 C0</td>
<td><code>xor eax, eax</code></td>
</tr>
<tr>
<td>F8</td>
<td><code>clc</code></td>
</tr>
<tr>
<td>1E C0</td>
<td><code>sbb eax, eax</code></td>
</tr>
<tr>
<td>25 56 34 42 24</td>
<td><code>and eax, 24423456h</code></td>
</tr>
<tr>
<td>25 28 48 21 42</td>
<td><code>and eax, 42214828h</code></td>
</tr>
<tr>
<td>6A 30</td>
<td><code>push 30h</code></td>
</tr>
<tr>
<td>58</td>
<td><code>pop eax</code></td>
</tr>
<tr>
<td>34 30</td>
<td><code>xor al, 30h</code></td>
</tr>
</tbody>
</table>

All bytes are printable
Various Ways to Set $eax$ to 0h

- **B8 00 00 00 00**  
  `mov eax, 0`  

- **33 C0**  
  `xor eax, eax`

- **F8**  
  `clc`

- **1E C0**  
  `sbb eax, eax`

- **25 56 34 42 24**  
  `and eax, 24423456h`

- **25 28 48 21 42**  
  `and eax, 42214828h`

- **6A 30**  
  `push 30h`

- **58**  
  `pop eax`

- **34 30**  
  `xor al, 30h`

All bytes that are ASCII letters are uppercase
Various Ways to Set $eax$ to $0h$

- **B8 00 00 00 00** \( \text{mov } eax, 0 \)
- **33 C0** \( \text{xor } eax, eax \)
- **F8** \( \text{clc} \)
- **1E C0** \( \text{sbb } eax, eax \)
- **25 56 34 42 24** \( \text{and } eax, 24423456h \)
- **25 28 48 21 42** \( \text{and } eax, 42214828h \)
- **6A 30** \( \text{push } 30h \)
- **58** \( \text{pop } eax \)
- **34 30** \( \text{xor } al, 30h \)

All bytes are alphanumeric
Overview
Existing Solution: Shellcode Encoders

- Begin with unencoded shellcode.

```plaintext
F7  44  A7  9F  C6  5E
43  AD  BA  38  81  27
F7  3F  10  EF  67  11
7B  F3  EB  B1  8A  16
A4  5F  41  D3  53  C9
ED  A6  2B  82  7A  A7
```
Overview
Existing Solution: Shellcode Encoders

- Begin with unencoded shellcode.
- Produce encoded shellcode (within encoding restriction).
Overview
Existing Solution: Shellcode Encoders

- Begin with unencoded shellcode.
- Produce encoded shellcode (within encoding restriction).
- Produce decoder (within encoding restriction).
Overview
Pros and Cons of Shellcode Encoders

Pros:
▶ It often works
▶ Can handle common cases automatically

Cons:
▶ Can expand the size of the shellcode, perhaps fatally
▶ Requires manual work to support new encodings
▶ Not guaranteed to work for an arbitrary encoding
▶ Generated code often has common sequences that can be detected by IDS signatures
▶ Encoder framework code is usually nasty
Synesthesia can act like a compiler that also inputs an encoding restriction.
Overview
Synesthesia: Re-Compiler Mode

- Synesthesia can input an existing code fragment, and find an equivalent version that also satisfies an encoding restriction.
Synesthesia can take existing binary shellcode blobs, and automatically encode them (and generate a decoder) to lie within the specified encoding restriction.
Overview

Synesthesia: Theoretical Properties

1. Fully automated, no manual analysis required
2. Static analysis, no dynamic analysis
   ▶ Don’t need access to a processor for the architecture
3. Flexible
   ▶ Supports arbitrary encoding restrictions
   ▶ Idea can be adapted to any processor
4. Exhaustive
   ▶ Guaranteed to find a solution within the encoding if one exists
   ▶ Can find all possible solutions
     ▶ Encompassing those instructions that are modelled
5. Optimal
   ▶ Can find the shortest solution (by # bytes or # instructions)
6. Metamorphic
   ▶ Can potentially produce self-modifying code
   ▶ May produce a different output every time
   ▶ Doesn’t use patterns or templates
     ▶ Hence no common byte sequences for IDS to catch
Overview

Synesthesia: Limitations

Synesthesia is **still a research idea** with practical limitations.

- Can be very expensive, especially for complex tasks.
  - Tends to work reasonably quickly for simple problems.
- Present implementation has limited support for synthesis of memory operations.
  - Discussed more thoroughly later.
- More research, and better SMT solvers are needed.
Symbolic Program Synthesis

Synthesizing C-Like Programs
Synthesizing Assembly Programs
Synthesizing Machine-Code Programs
Symbolic Program Synthesis
Motivating Example and Step #1: Enumerate Potential Solutions

Question: is it possible to create the function $x+1$ by using:

1. Two statements, where:
2. Each statement has one operator, and:
3. Each operator is $\neg$ (not) or $-$ (neg)?

We began by enumerating all possible programs:

\[
\begin{align*}
y &= \neg x; & y &= -x; \\
z &= \neg y; & z &= \neg y; \\
y &= \neg x; & y &= -x; \\
z &= -y; & z &= -y;
\end{align*}
\]
Symbolic Program Synthesis

Step #2: Encapsulate Variation into Components

We encapsulate all variation in the candidate programs using data items, called components.

\[ y = \neg x; \quad y = -x; \]
\[ z = \neg y; \quad z = \neg y; \]
\[ y = \neg x; \quad y = -x; \]
\[ z = -y; \quad z = -y; \]

\[ \text{COMPONENTS} = \langle \text{bool bop1, bool bop2} \rangle \]

- bool bop1: is first operation \( \neg \), or \( - \)?
- bool bop2: is second operation \( \neg \), or \( - \)?
Symbolic Program Synthesis

Step #3: Create Symbolic Representation in terms of Components

Describe all solutions with a **symbolic program** (using the components).

```c
bool bop1;  
bool bop2;  

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

Function $f$ takes one **input**: $\text{int } x$.

$$\text{INPUTS} = \langle \text{int } x \rangle$$

Question is now: can we set $bop1$ and $bop2$ so that $f(x) = x+1$ for all $x$?
We need to rephrase the question mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there values of $bop_1$, $bop_2$</td>
<td>...</td>
</tr>
</tbody>
</table>
Symbolic Program Synthesis
Step #4: Create Synthesis Formula

We need to rephrase the question mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there values of (bop_1, bop_2) (\exists bop_1, bop_2 \in \text{Bool} \cdot)</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Program Synthesis
Step #4: Create Synthesis Formula

We need to rephrase the question mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there values of ( bop1, bop2 ) ( \exists bop1, bop2 \in \text{Bool} \cdot )</td>
<td></td>
</tr>
<tr>
<td>Such that, for all values of ( x )</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Program Synthesis
Step #4: Create Synthesis Formula

★ We need to rephrase the question mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there values of ( bop1, bop2 ) such that, for all</td>
<td>( \exists \ bop1, bop2 \in \text{Bool} \cdot )</td>
</tr>
<tr>
<td>values of ( x )</td>
<td>( \forall x \in \text{BV}[32] \cdot )</td>
</tr>
</tbody>
</table>
We need to rephrase the question mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there values of bop1, bop2</td>
<td>$\exists \ bop1, \ bop2 \in \text{Bool}$ ·</td>
</tr>
<tr>
<td>Such that, for all values of x</td>
<td>$\forall \ x \in \text{BV}[32]$ ·</td>
</tr>
</tbody>
</table>

In the code

```plaintext
y = bop1 ? -x : ~x;
let y = bop1 ? -x : ~x in
z = bop2 ? -y : ~y;
let z = bop2 ? -y : ~y in
```
Symbolic Program Synthesis
Step #4: Create Synthesis Formula

▶ We need to rephrase the question mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Are there values of ( bop1, bop2 )</td>
<td>( \exists \ bop1, bop2 \in \text{Bool} )</td>
</tr>
<tr>
<td>Such that, for all values of ( x )</td>
<td>( \forall x \in \text{BV}[32] )</td>
</tr>
</tbody>
</table>

In the code

\[
\begin{align*}
y &= bop1 \ ? \ -x \ : \ \sim x; \\
z &= bop2 \ ? \ -y \ : \ \sim y;
\end{align*}
\]

let \( y = bop1 \ ? \ -x \ : \ \sim x \) in
let \( z = bop2 \ ? \ -y \ : \ \sim y \) in

\( z == x+1 \) is always true? \( z == x+1 \)
Create a synthesis formula consisting of four elements.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>∃COMPONENTS</td>
<td>Exists components</td>
<td>∃ bop1, bop2 ∈ Bool ·</td>
</tr>
<tr>
<td>∀INPUTS</td>
<td>For all inputs</td>
<td>∀ x ∈ BV[32] ·</td>
</tr>
<tr>
<td>φProgram</td>
<td>Program constraint</td>
<td>let y = bop1 ? -x : ~x in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>let z = bop2 ? -y : ~y in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z == x+1</td>
</tr>
<tr>
<td>φFunctionality</td>
<td>Functionality constraint</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic Program Synthesis

Step #5: Solve Synthesis Formula

Solve the synthesis formula with an SMT solver.

\[
\begin{aligned}
\exists \ bop_1, \ bop_2 \in \text{Bool} \cdot \\
\forall \ x \in \text{BV}[32] \cdot \\
\text{let } y = bop_1 \ ? \ -x : \sim x \ \text{in} \\
\text{let } z = bop_2 \ ? \ -y : \sim y \ \text{in} \\
z = x + 1
\end{aligned}
\]

\[
\begin{array}{c|c|c}
bop_1 & \rightarrow & \text{false} \\
bop_2 & \rightarrow & \text{true}
\end{array}
\]

Solution for \underline{COMPONENTS}

If the formula is unsolvable, the solver returns UNSAT.
Plug the solution for COMPONENTS into the symbolic program . . .

bool bop1; ▶
bool bop2; ▬

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

Solution for COMPONENTS

<table>
<thead>
<tr>
<th></th>
<th>◀</th>
<th>◆</th>
</tr>
</thead>
<tbody>
<tr>
<td>bop1</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>bop2</td>
<td>true</td>
<td>◆</td>
</tr>
</tbody>
</table>
Symbolic Program Synthesis  
Step #6: Interpret Synthesis Formula Solution

Plug the solution for \( \text{COMPONENTS} \) into the symbolic program ... 

\[
\text{int } f(\text{int } x) \\
\{ \\
\hspace{1em} \text{int } y = \neg x; \quad \textcolor{red}{\uparrow} \\
\hspace{1em} \text{int } z = -y; \quad \textcolor{blue}{\downarrow} \\
\hspace{1em} \text{return } z; \\
\}
\]

\[
\begin{array}{c|c}
\text{Solution for } \text{COMPONENTS} \\
\hline
\text{bop1} & \leftrightarrow \text{false} \quad \textcolor{red}{\uparrow} \\
\text{bop2} & \leftrightarrow \text{true} \quad \textcolor{blue}{\downarrow}
\end{array}
\]

... to obtain the desired program.
Symbolic Program Synthesis
More on Synthesis Formulas

Each formula in this presentation has roughly the same structure.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists$COMPONENTS $\forall$INPUTS $\phi_{\text{Program}}$ $\phi_{\text{Functionality}}$</td>
<td>Exists components For all inputs Program constraint Functionality constraint</td>
</tr>
</tbody>
</table>

This formula structure has several names in the literature:

- Exists/forall
- One quantifier alternation
- Effectively propositional
- Bernays-Schönfinkel
Symbolic Program Synthesis
Extending the Framework: More Operator Types

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

```c
char op1;

int f(int x)
{
    y = op1 == 0 ? -x :
        op1 == 1 ? ~x :
                x-1;
    return y;
}
```
Symbolic Program Synthesis
Extending the Framework: Unspecified Constants

We can extend the idea to incorporate **unspecified constants**:

```c
bool op;
char c; ▼

int f(char x)
{
    y = op ?
        x + c :
        x ^ c;

    return y;
}
```

Let's synthesize \( f(x) = \sim x \).

\[
\exists \langle \text{op} \in \text{Bool}, \ c \in \text{BV}[8] \rangle \cdot \\
\forall x \in \text{BV}[8] \cdot \text{let y = op ? x + c : x ^ c in y} = \sim x
\]

Constants are components, so solutions must include values for the constants.

Solution: \( \text{op} \mapsto \text{false}, \ c \mapsto 0xFF \). I.e. \( f(x) \) is \( x ^ \sim 0xFF \).
We can extend the idea to incorporate **unspecified constants**: 

```c
bool op;
char c;

int f(char x)
{
    y = op ? x + c : x ^ c;
    return y;
}
```

Let’s synthesize $f(x) = \lnot x$.

$$
\exists \langle op \in \text{Bool}, \downarrow c \in \text{BV}[8] \rangle. \quad \forall x \in \text{BV}[8].
$$

Let $y = op \ ? x + c : x \ ^\land c \ ^\land$ in

$y == \lnot x$

Constants are components, so solutions must include values for the constants.

Solution: $op \mapsto \false$, $c \mapsto 0xFF$.

* I.e. $f(x)$ is $x \ ^\land 0xFF$. 
We can extend the idea to incorporate **unspecified constants**:

```c
bool op;
char c;

int f(char x)
{
    y = op ?
        x + c :
        x ^ c;

    return y;
}
```

Let’s synthesize \( f(x) = \sim x \).

\[
\exists \langle \text{op} \in \text{Bool}, \text{c} \in \text{BV}[8] \rangle \cdot \\
\forall x \in \text{BV}[8] \cdot \\
\text{let } y = \text{op } ? x + c : x ^ c \text{ in } \\
y = \sim x
\]

**Constants are components**, so solutions **must** include values for the constants.

Solution: \( \text{op} \mapsto \text{false}, \text{c} \mapsto 0xFF \).

I.e. \( f(x) \) is \( x ^ 0xFF \).
Symbolic Program Synthesis
Synthesizing C-Like Programs
Synthesizing Assembly Programs
Synthesizing Machine-Code Programs
Synthesizing Assembly Programs

Plan for This Section

Roadmap for transitioning from C synthesis to ASM synthesis:

1. We define a simple assembly language, called SIMPLE.
   - Synesthesia also works for real assembly languages like X86.
   - However, X86 is more complex, and would not fit in an hour.
   - See source code for complete details on adapting to X86.

2. We devise a C representation for SIMPLE.
   - An enumeration for SIMPLE opcodes
   - A data structure for SIMPLE instructions
   - A data structure for SIMPLE machine states

3. We write a simulator for SIMPLE.
   - A function to update SIMPLE machine states
   - A function to simulate SIMPLE operations
   - A function to simulate SIMPLE instructions

4. We synthesize SIMPLE programs.
Synthesizing Assembly Programs

Transitioning from Synthesizing C Programs

### Synthesizing C Programs
```c
bool bop1;  
bool bop2;  

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

### Synthesizing ASM Programs
```assembly
Instruction i1;  
Instruction i2;  

state *f(state *in)  
{
    state *s1 = EmulateOne(in,i1);
    state *s2 = EmulateOne(s1,i2);
    return s2;
}
```

**Differences in synthesizing ASM programs versus C:**

- Components \(\heartsuit\) become **assembly language instructions**.
- Inputs and outputs \(\heartsuit\) become **machine states**.
  - Register and flag values, and/or memory contents.
Synthesizing Assembly Programs

SIMPLE Assembly Language

- SIMPLE has 8 32-bit registers, \( r_0 \) to \( r_7 \).
- Instructions are below; they work like you would expect.
  - \( rX \) and \( rY \) stand for any of the 32-bit registers.
  - \( \text{imm32} \) stands for any 32-bit constant value.

\[
\begin{align*}
\text{xor} & \quad rX, \quad rY & \quad \text{add} & \quad rX, \quad rY & \quad \text{mov} & \quad rX, \quad rY \\
\text{inc} & \quad rX & \quad \text{dec} & \quad rX & \quad \text{neg} & \quad rX & \quad \text{not} & \quad rX \\
\text{add} & \quad rX, \quad \text{imm32} & \quad \text{xor} & \quad rX, \quad \text{imm32} & \quad \text{and} & \quad rX, \quad \text{imm32} & \quad \text{or} & \quad rX, \quad \text{imm32}
\end{align*}
\]
enum Simple {
    XorRegReg,
    AddRegReg,
    MovRegReg,
    IncReg,
    DecReg,
    NegReg,
    NotReg,
    AddRegImm,
    XorRegImm,
    AndRegImm,
    OrRegImm,
};

We define an enumeration with one entry per instruction type.
We define a structure to represent instructions.

- **op**: mnemonic.
- **lhsRegNum**: left-hand-side register number.
- **rhsRegNum**: right-hand-side register number (if applicable).
- **imm32**: 32-bit constant value (if applicable).
We model the machine state as an array. E.g., `state[4]` is `r4`, etc.

```c
typedef state uint32[8];
```

Machine State

For more complex assembly languages, we’ll need (at least) flags and memory.
The function `Update(state *state, int regNum, uint32 value)`:

1. Copies an existing `state`;
2. Updates the value of register `regNum` to `value`;
3. Returns the new `state`.

Action of `Update(state, 3, 0)`

```
| r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 |
```

```
| r0 | r1 | r2 | 00 | r4 | r5 | r6 | r7 |
```

New Output State (Input State Copied, `r3` Updated)
state *Update(state *state, int regNum, uint32 value)

    state *out = new state;
    memcpy(out, state, sizeof(*state));
    out[regNum] = value;
    return out;
uint32 PerformOne(Simple op, uint32 l, uint32 r, uint32 i)

switch(op)
{
    case XorRegReg: return l ^ r; case AddRegReg: return l + r;
    case MovRegReg: return r;
    case IncReg: return l + 1; case DecReg: return l - 1;
    case NegReg: return -l; case NotReg: return ~l;
    case AddRegImm: return l + i; case XorRegImm: return l ^ i;
    case AndRegImm: return l & i; case OrRegImm: return l | i;
}

- Emulating SIMPLE is very easy.
The function *EmulateOne*:

1. Fetches the inputs from the *state*;
2. Performs the operation specified by instruction *i*;
3. Returns an updated state with the results of the instruction.
Synthesizing Assembly Programs
Comparison with Synthesizing C Programs

Synthesizing C Programs

```c
bool bop1; ▲
bool bop2; ▲

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

Synthesizing ASM Programs

```asm
Instruction i1; ▲
Instruction i2; ▲

state *f(state *in) ▲
{
    state *s1 = EmulateOne(in,i1);
    state *s2 = EmulateOne(s1,i2);
    return s2; ▲
}
```

Slide is duplicated from before. Now it should make sense.
To synthesize SIMPLE programs, we need to specify functionality constraints (input/output relationships) in terms of states:

\[
\phi_{\text{Functionality-Increment-} r_0} \\
\begin{align*}
    s_2[0] &= \text{in}[0]+1 ~ & \& \\
    s_2[1] &= \text{in}[1] ~ & \& \\
    s_2[2] &= \text{in}[2] ~ & \& \\
    s_2[3] &= \text{in}[3] ~ & \& \\
    s_2[4] &= \text{in}[4] ~ & \& \\
    s_2[5] &= \text{in}[5] ~ & \& \\
    s_2[6] &= \text{in}[6] ~ & \& \\
    s_2[7] &= \text{in}[7]
\end{align*}
\]

This constraint specifies: \( r_0 \) increments; other registers unchanged.
Our synthesis formula is:

$$\exists (i_1 \in \text{Instruction}, \ i_2 \in \text{Instruction}) \cdot \forall \text{in} \in \text{State} \cdot$$

let \( s_1 = \text{EmulateOne}(\text{in}, i_1) \) in

let \( s_2 = \text{EmulateOne}(s_1, i_2) \) in

$$\phi \text{Functionality-Increment-r0}$$
Our synthesis formula is:

\[ \exists \langle i_1 \in \text{Instruction}, i_2 \in \text{Instruction} \rangle \cdot \forall \text{in} \in \text{State} \cdot \]

let \( s_1 = \text{EmulateOne}(\text{in}, i_1) \) in

let \( s_2 = \text{EmulateOne}(s_1, i_2) \) in

\( \phi \) \text{Functionality-Increment-r0}

Solution:

\[
\begin{align*}
i_1 & \mapsto \{\text{AddRegImm}, 0, 0, 1\} \\
i_2 & \mapsto \{\text{OrRegImm}, 0, 0, 0\}
\end{align*}
\]

I.e.:

\[
\begin{align*}
\text{add } r0, 1 \\
\text{or } r0, 0
\end{align*}
\]
Synthesizing Assembly Programs

Obtaining Alternative Solutions

Let's say we want a solution different from:

\[
\text{add } r0, 1 \\
or r0, 0
\]

Existing synthesis formula:

\[
\exists \langle i_1 \in \text{Instruction}, i_2 \in \text{Instruction} \rangle \cdot \\
\forall in \in \text{State} \cdot \\
\text{let } s_1 = \text{EmulateOne}(in, i_1) \text{ in} \\
\text{let } s_2 = \text{EmulateOne}(s_1, i_2) \text{ in} \\
\phi \text{Functionality-Increment-r0}
\]
Synthesizing Assembly Programs

Obtaining Alternative Solutions

Let’s say we want a solution different from:

\[
\begin{align*}
\text{add } r0, 1 \\
\text{or } r0, 0
\end{align*}
\]

Existing synthesis formula:

\[
\exists \langle i1 \in \text{Instruction}, i2 \in \text{Instruction} \rangle \cdot
\forall in \in \text{State} \cdot
\text{let } s1 = \text{EmulateOne}(in,i1) \text{ in }
\text{let } s2 = \text{EmulateOne}(s1,i2) \text{ in }
\phi \text{Functionality-Increment-r0}
\]

Add these new terms:

\[
\begin{align*}
i1.\text{op} &\neq \text{AddRegImm} \mid i1.\text{lhsRegNum} \neq 0 \mid i1.\text{imm32} \neq 1 \\
i2.\text{op} &\neq \text{OrRegImm} \mid i2.\text{lhsRegNum} \neq 0 \mid i2.\text{imm32} \neq 0
\end{align*}
\]
The first 8 solutions with 2 instructions for \( r_0 = r_0 + 1 \):

<table>
<thead>
<tr>
<th>add r0, 1</th>
<th>mov r2, r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>or r0, 0</td>
<td>inc r0</td>
</tr>
<tr>
<td>dec r0</td>
<td>xor r0, 0</td>
</tr>
<tr>
<td>add r0, 2</td>
<td>add r0, 1</td>
</tr>
<tr>
<td>not r0</td>
<td>mov r0, r0</td>
</tr>
<tr>
<td>neg r0</td>
<td>inc r0</td>
</tr>
<tr>
<td>inc r0</td>
<td>add r0, 20D910C5h</td>
</tr>
<tr>
<td>mov r0, r0</td>
<td>add r0, 0DF26EF3Ch</td>
</tr>
</tbody>
</table>

Some solutions have **NOP** instructions in them.
Synthesizing Assembly Programs

Variable-Length Programs

Instruction i1, i2, i3, i4, i5;
int numInstrs;

state *f(state *in) {
    state *s1 = EmulateOne(in, i1);
    state *s2 = EmulateOne(s1, i2);
    state *s3 = EmulateOne(s2, i3);
    state *s4 = EmulateOne(s3, i4);
    state *s5 = EmulateOne(s4, i5);

    return numInstrs == 1 ? s1 :
        numInstrs == 2 ? s2 :
        numInstrs == 3 ? s3 :
        numInstrs == 4 ? s4 :
                       s5;
}

So far, our formulas used a fixed number of instructions.
We can easily extend to “up to” a fixed number, as shown.
The value of numInstrs in the solution tells us how many instructions were used.

We could revert to the prior behavior by adding a constraint: numInstrs == 2.
Or, a range of lengths:
2 <= numInstrs <= 4.
Symbolic Program Synthesis
    Synthesizing C-Like Programs
    Synthesizing Assembly Programs
    Synthesizing Machine-Code Programs
Synthesizing Machine-Code Programs

Plan for This Section

Roadmap for transitioning from ASM synthesis to machine code:

1. Define a machine code encoding for SIMPLE: SIMPLEMC.
2. Write a disassembler for SIMPLEMC into Instruction objects.
   - A function to decode SIMPLE opcodes from SIMPLEMC
   - A function to decode whole SIMPLE instructions
3. Use the existing SIMPLE machinery to synthesize SIMPLEMC.
Synthesizing Machine-Code Programs

Transitioning from Synthesizing ASM Programs

Synthesizing SIMPLE Programs

```c
Instruction i1, i2;

state *f(state *in)
{
    state *s1=EmulateOne(in,i1);
    state *s2=EmulateOne(s1,i2);
    return s2;
}
```

Synthesizing SIMPLEMC Programs

```c
char mc[256];

state *f(state *in)
{
    int l1, l2;
    Instruction i1, i2;
    Decode(mc, 0, &l1, &i1);
    Decode(mc, l1, &l2, &i2);
    state *s1=EmulateOne(in,i1);
    state *s2=EmulateOne(s1,i2);
    return s2;
}
```

Differences in synthesizing ASM programs versus machine code:

- Components become machine code bytes.
- Machine-code formula must decode instructions.
### Synthesizing Machine-Code Programs

SIMPLE Machine Language

Machine-code encoding for binary `reg/reg` SIMPLE instructions.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>00 000 001</td>
<td>01 xor r0, r1</td>
</tr>
<tr>
<td>53</td>
<td>01 010 011</td>
<td>53 add r2, r3</td>
</tr>
<tr>
<td>A5</td>
<td>10 100 101</td>
<td>A5 mov r4, r5</td>
</tr>
</tbody>
</table>

**Opcode** is 00: xor; 01: add; 10: mov. Register #s in lower fields.
## Synthesizing Machine-Code Programs

**SIMPLE Machine Language**

Machine-code encoding for unary `reg` SIMPLE instructions.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Reg/Op</th>
<th>Reg</th>
<th>Instruction</th>
<th>Opcode</th>
<th>Reg/Op</th>
<th>Reg</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>000</td>
<td>C4</td>
<td><code>inc r4</code></td>
<td>101</td>
<td>001</td>
<td>CD</td>
<td><code>dec r5</code></td>
</tr>
<tr>
<td>010</td>
<td>110</td>
<td>D6</td>
<td><code>neg r6</code></td>
<td>111</td>
<td>011</td>
<td>DF</td>
<td><code>not r7</code></td>
</tr>
</tbody>
</table>

Opcode field is 11.

*Middle 3 are 000: inc; 001: dec; 010: neg; 011: not.*

Register number in lowest field.
Synthesizing Machine-Code Programs
SIMPLE Machine Language

Machine-code encoding for binary \texttt{reg/imm32} SIMPLE instructions.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Reg/Op</th>
<th>Reg</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0 00 11 100 000</td>
<td>E0 78 56 34 12</td>
<td>add r0, 12345678h</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E9 00 11 101 001</td>
<td>E9 12 34 56 78</td>
<td>xor r1, 78563412h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Opcode field is 11.

Middle 3 are 100: \texttt{add}; 101: \texttt{xor}; 110: \texttt{and}; 111: \texttt{or}.

Register number in lowest field. Constant follows opcode byte.
Simple DecodeOpcode(int firstByte)

int topTwo = (firstByte>>6) & 3;
int midThree = (firstByte>>3) & 7;

if(topTwo != 0b11)
    return XorRegReg + topTwo;

return IncReg + midThree;

This function decodes the opcode from a SIMPLEMC byte.
void Decode(char *bytes, int eip, int *length, Instruction *ins)

  ▶ ins->op = DecodeOpcode(bytes[eip]);
  ▶ *length = ins->op < AddRegImm ? 1 : 5;
  ▶ ins->lhsRegNum = (firstByte>>3) & 7;
  ▶ ins->rhsRegNum = firstByte & 7;
  ▶ if(ins->op > MovRegReg)
    ▶ ins->lhsRegNum = ins->rhsRegNum;
  ▶ ins->imm32 = *(uint32 *)&bytes[eip+1]);

This function decodes an instruction from SIMPLEMC bytes.

1. Fetch the opcode ◀.
2. Determine the instruction’s length ◀.
3. Extract the register numbers ◀.
4. Extract the 32-bit constant ◀.
Synthesizing Machine-Code Programs

Comparison with Synthesizing ASM Programs

Synthesizing SIMPLE Programs

Instruction i1, i2;

state *f(state *in)
{

state *s1=EmulateOne(in,i1);
state *s2=EmulateOne(s1,i2);
return s2;
}

Slide is duplicated from before. Now it should make sense.
Extensions

Encoding Restrictions
Synthesis of Equivalent Snippets
Finding the Shortest Program
Synthesizing Decoders
Input State Preconditions
Integration with Exploit Generation
Encoding Restrictions

Overview

Recall from before that our formulas have roughly this structure.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>∃COMPONENTS</td>
<td>Exists components</td>
<td>( \exists \text{mc} \in \text{Array[BV[8] → BV[8]]} )</td>
</tr>
<tr>
<td>∀INPUTS</td>
<td>For all inputs</td>
<td>( \forall \text{in} \in \text{State} )</td>
</tr>
<tr>
<td>( \phi \text{Program} )</td>
<td>Program constraint</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \phi \text{Functionality} )</td>
<td>Functionality constraint</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
Recall from before that our formulas have roughly this structure.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\exists$</td>
<td>COMPONENTS</td>
<td>$\exists \ mc \in \text{Array}[\text{BV}[8] \to \text{BV}[8]]$</td>
</tr>
<tr>
<td>$\forall$</td>
<td>INPUTS</td>
<td>$\forall \ in \in \text{State}$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>Program</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>Functionality</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>Encoding</td>
<td>Encoding constraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Shown next</em></td>
</tr>
</tbody>
</table>

We begin by adding **encoding restrictions**; say, no NULL bytes.
Let’s say we don’t want any NULL bytes in our machine code. We need to phrase that property mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all array indices $i$</td>
<td>$\forall i \cdot \text{mc}[i] \neq 0x00$</td>
</tr>
</tbody>
</table>
Encoding Restrictions

Example: No NULL Bytes

- Let’s say we don’t want any NULL bytes in our machine code.
- We need to phrase that property mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all array indices</td>
<td>$\forall i \in BV[8] \cdot$</td>
</tr>
</tbody>
</table>

Within our synthesized machine code

$i < \text{len1} + \text{len2} \Rightarrow \text{mc}[i] \neq 0x00$

$\forall i \in BV[8] \cdot i < \text{len1} + \text{len2} \Rightarrow \text{mc}[i] \neq 0x00$
Encoding Restrictions
Example: No NULL Bytes

▶ Let’s say we don’t want any NULL bytes in our machine code.
▶ We need to phrase that property mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all array indices ( i )</td>
<td>( \forall i \in \text{BV}[8] ).</td>
</tr>
<tr>
<td>Within our synthesized machine code</td>
<td></td>
</tr>
</tbody>
</table>
Let’s say we don’t want any NULL bytes in our machine code.

We need to phrase that property mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all array indices $i$</td>
<td>$\forall i \in \text{BV}[8]$.</td>
</tr>
<tr>
<td>Within our synthesized machine code</td>
<td>$i &lt; \text{len1} + \text{len2} \Rightarrow$</td>
</tr>
</tbody>
</table>
Encoding Restrictions
Example: No NULL Bytes

- Let’s say we don’t want any NULL bytes in our machine code.
- We need to phrase that property mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all array indices ( i )</td>
<td>( \forall i \in \text{BV}[8]. )</td>
</tr>
<tr>
<td>Within our synthesized machine code</td>
<td>( i &lt; \text{len1} + \text{len2} ) ⇒</td>
</tr>
<tr>
<td>Machine code byte ( #i ) is not ( 0x00 )</td>
<td>( )</td>
</tr>
</tbody>
</table>
Let’s say we don’t want any NULL bytes in our machine code.

We need to phrase that property mathematically:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>For all array indices $i$</td>
<td>$\forall i \in \text{BV}[8]$.</td>
</tr>
<tr>
<td>Within our synthesized machine code</td>
<td>$i &lt; \text{len1} + \text{len2} \Rightarrow$</td>
</tr>
<tr>
<td>Machine code byte $#i$ is not 0x00</td>
<td>$\text{mc}[i] \neq 0x00$</td>
</tr>
</tbody>
</table>

$\phi_{\text{Non-NULL}} := [\forall i : \text{BV}[8] \cdot i < \text{len1} + \text{len2} \Rightarrow \text{mc}[i] \neq 0x00]$
These examples are all $\forall i : \text{BV}[8] \cdot i < \text{len1} + \text{len2} \Rightarrow \phi_{\text{Byte}}$, where $\phi_{\text{Byte}}$ is · · ·:

<table>
<thead>
<tr>
<th>Encoding Restriction</th>
<th>$\phi_{\text{Byte}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No NULL Bytes</td>
<td>mc[i] $\neq$ 0x00</td>
</tr>
<tr>
<td>No ’%’ Bytes</td>
<td>mc[i] $\neq$ 0x25</td>
</tr>
<tr>
<td>All ASCII are Uppercase</td>
<td>$\neg (mc[i] \geq 0x61 \land mc[i] \leq 0x7A)$</td>
</tr>
<tr>
<td>All Bytes Printable</td>
<td>$(mc[i] \geq 0x21 \land mc[i] \leq 0x7F)$</td>
</tr>
<tr>
<td>All Bytes Alphanumeric</td>
<td>$(mc[i] \geq 0x30 \land mc[i] \leq 0x39) \lor (mc[i] \geq 0x41 \land mc[i] \leq 0x5A) \lor (mc[i] \geq 0x61 \land mc[i] \leq 0x7A)$</td>
</tr>
</tbody>
</table>
Encoding Restrictions
More Complex Examples: Bytes Must Increase

Let’s say our shellcode bytes must monotonically increase.
Encoding Restrictions
More Complex Examples: Bytes Must Increase

Let's say our shellcode bytes must monotonically increase.

\[ \forall i : \text{BV}[8] \cdot i < \text{len1} + \text{len2} - 1 \Rightarrow (mc[i] \leq mc[i+1]) \]
Encoding Restrictions

More Complex Examples: Bytes Must Increase

Let’s say our shellcode bytes must monotonically increase.

\[ \forall i : \text{BV}[8] \cdot i < \text{len}1 + \text{len}2 - 1 \Rightarrow (\text{mc}[i] \leq \text{mc}[i+1]) \]

Let’s say our shellcode bytes must strictly increase.
Let’s say our shellcode bytes must monotonically increase.

\[ \forall i : BV[8] \cdot i < len1 + len2 - 1 \Rightarrow (mc[i] \leq mc[i+1]) \]

Let’s say our shellcode bytes must strictly increase.

\[ \forall i : BV[8] \cdot i < len1 + len2 - 1 \Rightarrow (mc[i] < mc[i+1]) \]
Let’s say we don’t want any repeated bytes in our shellcode.
Encoding Restrictions

More Complex Examples: No Duplicate Bytes

Let’s say we don’t want any repeated bytes in our shellcode.

$$\forall i : \text{BV}[8], j : \text{BV}[8] \cdot i < \text{len}1 + \text{len}2 - 1 \land j < i \Rightarrow (mc[j] \neq mc[i])$$
Let’s say our shellcode must alternate between even and odd bytes.
Let’s say our shellcode must alternate between even and odd bytes.

\[ \forall i : \text{BV}[32] \cdot i < \text{len1} + \text{len2} - 1 \Rightarrow (mc[i] ^ mc[i+1]) \& 1 == 1 \]
Let’s say our shellcode must alternate between even and odd bytes.

\[ \forall i : BV[32] \cdot i < len1 + len2 - 1 \Rightarrow (mc[i] \land mc[i+1]) \& 1 == 1 \]

... and, additionally, the first byte is even:
Let’s say our shellcode must alternate between even and odd bytes.

\[ \forall i : \text{BV}[32] \cdot i < \text{len1} + \text{len2} - 1 \Rightarrow (mc[i] \oplus mc[i+1]) \& 1 = 1 \]

... and, additionally, the first byte is even:

\[ mc[0] \& 1 = 0 \]
A word \( w \) is prime if only 1 and \( w \) divides evenly into it. I.e.:

\[
\forall d : \text{BV}[16] \cdot (2 \leq d \&\& d < w) \implies w \mod d \neq 0
\]
Encoding Restrictions

Solutions

Interesting solutions to $\text{eax} == 0x0$

First byte of each instruction $< 0x20$, all bytes non-NULL

0D CA 01 4B FE  or eax, 0FE4B01CAh
1D CA 16 B3 5A  sbb eax, 5AB316CAh
19 C0  sbb eax, eax

Interesting solutions to $\text{eax} == 0x12345678$

Alternating even/odd

B8 7B 56 35 7A  mov eax, 7A35567Bh
25 FC F7 34 17  and eax, 1734F7FCh

Alternating even/odd (first byte odd)

81 C8 7D 56 B5 92  or eax, 92B5567Dh
81 E0 7D 56 FD 92  and eax, 92FD567Dh
25 78 57 36 13  and eax, 13365778h
Encoding Restrictions

In General

- This scheme is compatible with **absolutely any** encoding restriction that can be expressed as a first-order formula.
- This does imply that we know what the encoding restrictions are and can express them as a formula.
- Later, we’ll see we can automatically determine this and not explicitly model it.
Extensions

Encoding Restrictions
Synthesis of Equivalent Snippets
Finding the Shortest Program
Synthesizing Decoders
Input State Preconditions
Integration with Exploit Generation
Synthesis of Equivalent Snippets

Conceptually

- Suppose we already have machine code that does what we want, but it doesn’t satisfy the encoding restrictions.
- Simply express the input/output behavior of that code, and use that as the functional constraint.
Synthesis of Equivalent Snippets

Step #1: Assemble the Code

First, assemble the code you wish to replace.
Synthesis of Equivalent Snippets

Step #2: Synthesize Replacement

Next, synthesize equivalent code within the encoding.
If desired, omit irrelevant registers or flags from the functionality constraint.
Extensions

Encoding Restrictions
Synthesis of Equivalent Snippets
Finding the Shortest Program
Synthesizing Decoders
Input State Preconditions
Integration with Exploit Generation
Finding the Shortest Program

Conceptually

- Write $\text{Good}(\text{COMPONENTS})$ if the program defined by $\text{COMPONENTS}$ satisfies all functional and encoding constraints.
- Now our question is: what is the shortest good program?

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a best program</td>
<td>$\exists\text{BEST}$</td>
</tr>
</tbody>
</table>
Finding the Shortest Program

Conceptually

- Write $\text{Good}(\text{COMPONENTS})$ if the program defined by $\text{COMPONENTS}$ satisfies all functional and encoding constraints.

- Now our question is: what is the shortest good program?

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a best program</td>
<td>$\exists \text{BEST}$</td>
</tr>
<tr>
<td>Which is good, and</td>
<td>$\text{Good}(\text{BEST})$</td>
</tr>
</tbody>
</table>
Finding the Shortest Program

Conceptually

- Write $\textit{Good}(\text{COMPONENTS})$ if the program defined by \text{COMPONENTS} satisfies all functional and encoding constraints.
- Now our question is: what is the shortest \textit{good} program?

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a best program</td>
<td>$\exists \text{BEST}$</td>
</tr>
<tr>
<td>Which is good, and</td>
<td>$\textit{Good}(\text{BEST})$</td>
</tr>
<tr>
<td>For all other programs</td>
<td>$\forall \text{OTHER}$</td>
</tr>
</tbody>
</table>
Finding the Shortest Program

Conceptually

- Write $\text{Good}(\text{COMPONENTS})$ if the program defined by $\text{COMPONENTS}$ satisfies all functional and encoding constraints.
- Now our question is: what is the shortest good program?

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a best program?</td>
<td>$\exists \text{BEST}$</td>
</tr>
<tr>
<td>Which is good, and</td>
<td>$\text{Good}(\text{BEST})$</td>
</tr>
<tr>
<td>For all other programs</td>
<td>$\forall \text{OTHER}$</td>
</tr>
<tr>
<td>If the other program is good</td>
<td>$\text{Good}(\text{OTHER})$</td>
</tr>
</tbody>
</table>
Finding the Shortest Program

Conceptually

- Write $Good(COMPONENTS)$ if the program defined by $COMPONENTS$ satisfies all functional and encoding constraints.
- Now our question is: what is the shortest good program?

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a best program</td>
<td>$\exists BEST$</td>
</tr>
<tr>
<td>Which is good, and</td>
<td>$Good(BEST)$</td>
</tr>
<tr>
<td>For all other programs</td>
<td>$\forall OTHER$</td>
</tr>
<tr>
<td>If the other program is good</td>
<td>$Good(OTHER)$</td>
</tr>
<tr>
<td>The other is at least as long</td>
<td>$\Rightarrow \text{Length}(BEST) \leq \text{Length}(OTHER)$</td>
</tr>
</tbody>
</table>

- Here we quantify over all solutions.
Finding the Shortest Program

Conceptually

Here is a shortest solution to \( \phi_{\text{Functionality-Increment-} r0} \):

```
AD   mov r5, r5
CO   inc r0
```
Finding the Shortest Program

Conceptually

Here is a shortest solution to $\phi_{\text{Functionality-Increment-}r_0}$:

```
AD mov r5, r5
C0 inc r0
```

If we wanted a longest solution, we could change our condition to $\text{Length(BEST)} \geq \text{Length(OTHER)}$.

```
D0 01 00 00 00 add r0, 1
F8 00 00 00 00 00 or r0, 0
```
Extensions

- Encoding Restrictions
- Synthesis of Equivalent Snippets
- Finding the Shortest Program

**Synthesizing Decoders**
- Input State Preconditions
- Integration with Exploit Generation
We shall now automate synthesis of decoder loops.

```plaintext
; Initialize counter
@loop:
; Get encoded byte
; Decode encoded byte
; Store decoded byte
; Decrement counter
; Loop if counter non-zero
```

How do we specify functional constraints for a loop?

- **Loop invariants**: the “work” done by an iteration.
- **Loop variants**: proving that the loop terminates.
Let’s review loop invariants and variants with an example.
We show that max terminates with the greatest element of arr.

```c
int max(int *arr, int len)
{
    assert(len > 0);
    int m = arr[0];

    for(int i=1;i<len;++i)
        if(arr[i] > m)
            m = arr[i];

    return m;
}
```

Validate input length.
Set m to first element.
Loop through array:
Is current element bigger?
If so, save it.
Return largest element.
Synthesizing Decoders

Loop Invariants

- A loop invariant says that:
  - If, before an iteration, some statement is true
  - Then, after the iteration, the statement is still true.
Synthesizing Decoders

Loop Invariants

- A loop invariant says that:
  - If, before an iteration, some statement is true
  - Then, after the iteration, the statement is still true.

```java
for(int i=1; i<len; ++i)
    if(arr[i] > m)
        m = arr[i];
```

- Our loop invariant is that:
  - If, before iteration #j, \( m = \text{MAX}(\text{arr}, 0, j-1) \), then:

```
Iteration j-1: m=MAX(arr)
```

```
0  ...  j-1  j  ...  len-1
```
A loop invariant says that:

- If, before an iteration, some statement is true
- Then, after the iteration, the statement is still true.

```java
for(int i=1; i<len; ++i)
    if(arr[i] > m)
        m = arr[i];
```

Our loop invariant is that:

- If, before iteration \( \# j \), \( m = \text{MAX}(arr,0,j-1) \), then:
- After iteration \( \# j \), \( m = \text{MAX}(arr,0,j) \).

<table>
<thead>
<tr>
<th>0</th>
<th>⋯</th>
<th>j-1</th>
<th>j</th>
<th>⋯</th>
<th>len-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration ( j ): ( m = \text{MAX}(arr) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synthesizing Decoders
Loop Invariants

- Our loop invariant is that:
  - If, before iteration \( j \), \( m = \text{MAX}(arr,0,j-1) \), then:
  - After iteration \( j \), \( m = \text{MAX}(arr,0,j) \).

```java
for(int i=1;i<len;++i)
    if(arr[i] > m)
        m = arr[i];
```

- The loop invariant is true because:

  - If \( arr[j] \leq m \), then: \( arr[j] \) is not greater than some previous element (\( m \)).
    - Thus the existing \( m = \text{MAX}(arr,0,j-1) = \text{MAX}(arr,0,j) \).
  - If \( arr[j] > m \), then: \( arr[j] \) is greater than all previous elements.
    - Thus \( arr[j] = \text{MAX}(arr,0,j) \), and \( m \) becomes \( arr[j] \).
Synthesizing Decoders

Loop Invariants

- Our loop invariant is that:
  - If, before iteration \( #j \), \( m = \text{MAX}(\text{arr},0,j-1) \), then:
  - After iteration \( #j \), \( m = \text{MAX}(\text{arr},0,j) \).

```java
for(int i=1;i<len;++i)
    if(arr[i] > m)
        m = arr[i];
```

- The loop invariant is true because:
  - If \( \text{arr}[j] \) \(<=\) \( m \), then:
    - \( \text{arr}[j] \) is not greater than some previous element (\( m \)).
    - Thus the existing \( m = \text{MAX}(\text{arr},0,j-1) = \text{MAX}(\text{arr},0,j) \).
Synthesizing Decoders

Loop Invariants

Our loop invariant is that:

- If, before iteration \#j, \( m = \text{MAX}(\text{arr},0,j-1) \), then:
- After iteration \#j, \( m = \text{MAX}(\text{arr},0,j) \).

```java
for(int i=1;i<len;++i)
    if(arr[i] > m)
        m = arr[i];
```

The loop invariant is true because:

- If \( \text{arr}[j] \leq m \), then:
  - \( \text{arr}[j] \) is not greater than some previous element (\( m \)).
  - Thus the existing \( m = \text{MAX}(\text{arr},0,j-1) = \text{MAX}(\text{arr},0,j) \).

- If \( \text{arr}[j] > m \), then:
  - \( \text{arr}[j] \) is greater than all previous elements.
  - Thus \( \text{arr}[j] \) is \( \text{MAX}(\text{arr},0,j) \), and \( m \) becomes \( \text{arr}[j] \).
1 int m = arr[0];
2 for(int i=1;i<len;++i)
3   if(arr[i] > m)
4     m = arr[i];
5 return m;

▶ Because of line #1, before iteration #1, m = MAX(arr,0,0).
▶ After every iteration #j, m = MAX(arr,0,j).
   ▶ Specifically, after iteration #len-1, m = MAX(arr).
▶ Thus the code above correctly computes the array maximum.
   ▶ “Partially correct” because we have not yet proved termination.
A loop variant says that:

- If, before an iteration, the “amount of work remaining” is $n$,
- Then, after the iteration, “amount of work remaining” is $<n$.
- The “amount of work remaining” cannot decrease forever.

The variant function gives the amount of work remaining.
Define a variant \( v(i) = \text{len}-i \) (number of iterations left).

<table>
<thead>
<tr>
<th>#</th>
<th>Code</th>
<th>i</th>
<th>v(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>for(int i=1;i&lt;len;++i)</code></td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td><code>if(arr[i] &gt; m)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>m = arr[i];</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming \( \text{len} = 10 \)
Define a variant $v(i) = \text{len} - i$ (number of iterations left).

<table>
<thead>
<tr>
<th>#</th>
<th>Code</th>
<th>$i$</th>
<th>$v(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for(int $i=1; i&lt;\text{len}; ++i$)</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>if($\text{arr}[i] &gt; m$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$m = \text{arr}[i]$;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming $\text{len} = 10$
Define a variant \( v(i) = \text{len} - i \) (number of iterations left).

<table>
<thead>
<tr>
<th>#</th>
<th>Code</th>
<th>i</th>
<th>v(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>for(int i=1; i&lt;len; ++i)</code></td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td><code>if(arr[i] &gt; m)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>m = arr[i];</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming \( \text{len} = 10 \)
Define a variant $v(i) = \text{len}-i$ (number of iterations left).

<table>
<thead>
<tr>
<th>#</th>
<th>Code</th>
<th>i</th>
<th>$v(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>for(int i=1;i&lt;len;++i)</code></td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td><code>if(arr[i] &gt; m)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>m = arr[i];</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assuming len = 10

When $i=\text{len}$ ($v(i)=0$), the loop terminates.
Synthesizing Decoders
Loop Variants

```java
for(int i=1; i<len; ++i)
if(arr[i] > m)
m = arr[i];
```

- Properties of variant $v(i) = len - i$:
  - For every iteration, $v(i) \geq 0$ (non-negative, minimum 0).
  - For each iteration $\#i$, $v(i+1) < v(i)$ (it decreases).
    - Because $i$ increases on line $\#1$.
  - $v(i)$ can only decrease $len - 1$ times (descent is finite).
  - When $v(i)$ becomes 0, the loop terminates.

- Therefore, the loop executes a finite number of times ($len - 1$).
- Therefore, the loop terminates.
  - “Total correctness”: partial correctness plus termination.
If we synthesize code with the properties below, it is guaranteed to be a terminating loop that decodes the shellcode. As a bonus, it encodes the shellcode automatically.

**Loop variant:** \( v(i) = \text{len} - i \).

- Also, before iteration \( \neq 0 \), some register \( rC \) is set to \( \text{len}=v(0) \).

**Loop invariant:**

- If, before iteration \( i \), bytes \( 0..i-1 \) are decoded,
- Then after iteration \( i \):
  1. Bytes \( 0..i \) are decoded.
  2. \( rC \) has decreased by one.
  3. The new \( eip \) is either:
     - The beginning of the loop (if \( rC \neq 0 \))
     - The instruction after the loop (if \( rC \equiv 0 \))
## Synthesizing Decoders

### Changes to SIMPLE's Machine State

The new machine state model contains:

1. The registers r0…r7, as before;
2. The program counter, eip;
3. The current shellcode contents sc;
4. The current shellcode pointer scptr.

### New SIMPLE Machine State

<table>
<thead>
<tr>
<th>Registers</th>
<th>eip</th>
<th>Shellcode: Bytes sc</th>
<th>Pointer scptr</th>
</tr>
</thead>
</table>
### Synthesizing Decoders

**Changes to SIMPLE**

---

<table>
<thead>
<tr>
<th>Instructions Added to SIMPLE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>getscbyte rX \ rX = sc[scptr]</td>
</tr>
<tr>
<td>putscbyte rX \ sc[scptr++] = rX</td>
</tr>
<tr>
<td>jnz rX, imm8 \ If rX ≠ 0, jump to eip–imm8</td>
</tr>
</tbody>
</table>

- Added instructions to get and set shellcode bytes.
- Added a `jnz` instruction.

---

<table>
<thead>
<tr>
<th>Instructions Removed from SIMPLE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc rX \ dec rX \ or rX, imm32</td>
</tr>
</tbody>
</table>

- Removed a few arithmetic instructions.
  - They didn’t fit in the instruction set anymore.
We synthesize two programs simultaneously:

1. Initialization
   - Sets some register to the shellcode length

2. The loop body
   - Decrypts a shellcode byte
   - Leaves all other shellcode bytes in tact
   - Decreases the counter
   - Branches back to the top of the loop if counter non-zero
Synthesizing Decoders

Decoder Skeleton: Initialization

Initialize

Loop Body

Done

Synthesis formula for initialization block:

<table>
<thead>
<tr>
<th>English</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some register ( rC ) contains the length of</td>
<td>( \exists , \text{ctr} \in \text{BV}[3] )</td>
</tr>
<tr>
<td>the shellcode</td>
<td>( \text{stateAfter}[\text{regs}[\text{ctr}]] == \text{len} )</td>
</tr>
</tbody>
</table>
Synthesizing Decoders
Decoder Skeleton: Loop Body

Synthesis formula for loop body:

If, before iteration:

- \( \text{scptr is within sc} \) \hspace{1cm} \text{scptr} < \text{len}-1 \hspace{1cm} \& \&
- \( \text{sc[0...scptr-1]} \)
- \( \forall i \in \text{BV[32]} \)
- \( \text{is decoded} \)
- \( i < \text{scptr} \rightarrow \)
- \( \text{sc[i]} == \text{origsc[i]} \)

Then, after iteration:

- \( \text{rC decrements} \) \hspace{1cm} \text{rC}\text{After} == \text{rC}-1
- \( \text{scptr increments} \) \hspace{1cm} \text{sc}\text{ptr}\text{After} == \text{scptr}+1
- \( \forall i \in \text{BV[32]} \)
- \( i \leq \text{scptr} \rightarrow \)
- \( \text{sc}\text{After[i]} == \text{origsc[i]} \)
- \( \text{If } \text{rC} \neq 0 \) \hspace{1cm} \text{eip}\text{After} == \text{rC}\text{After} \neq 0 ?
- \( \text{Loop again} \)
- \( \text{Otherwise terminate} \)

\( \text{@loop_body} : \)

\( \text{@done} \)
Synthesizing Decoders

Decoder Skeleton

To make the problem more tractable, we can break the body up into blocks, and specify their functional constraints individually.

This excludes solutions where parts are interleaved.
Here is a decoder synthesized with no encoding restrictions.

- Not very interesting.
Synthesizing Decoders

Synthesized Decoder: Some Bytes Randomly Disallowed

- F5 40 00 00 00 mov r5, 40h
- @here:
  - C3 getscbyte r3
  - E5 FF FF FF FF add r5, 0FFFFFFFFh
  - D3 neg r3
  - CB putscbyte r3
  - FB FD jnz r5, @here

- Some bytes in the shellcode were randomly disallowed.
  - Applying neg to each byte bypassed the restriction.
Synthesizing Decoders

Synthesized Decoder: Encode to Printable Characters

This example encodes each byte using two printable bytes.

```
    mov r7, 40h
@here:
    xor r5, 000000A4h
    getscbyte r0
    mov r2, 80008081h
    getscbyte r1
    add r1, r1
    add r0, r1
    putscbyte r0
    add r7, 0FFFFFFFFh
    jnz r7, @here
```

Decoded/Encoded

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>50</td>
<td>28</td>
</tr>
<tr>
<td>8C</td>
<td>4A</td>
<td>21</td>
</tr>
<tr>
<td>1C</td>
<td>32</td>
<td>75</td>
</tr>
<tr>
<td>29</td>
<td>3D</td>
<td>76</td>
</tr>
</tbody>
</table>

NOP instructions are in red.
Synthesizing Decoders

Synthesized Decoder: Encode to ASCII Alphanumeric

This example encodes each byte using two alphanumerical bytes.

```assembly
mov r6, 40h
@here:
    getscbyte r4
    add r4, 43D087B6h
    mov r2, r4
    add r2, r2
    getscbyte r0
    xor r2, r0
    putscbyte r2
    add r6, 0FFFFFFFFh
    jnz r6, @here
```

Decoded/Encoded

<table>
<thead>
<tr>
<th></th>
<th>30</th>
<th>6C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8C</td>
<td>38</td>
<td>50</td>
</tr>
<tr>
<td>1C</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>29</td>
<td>??</td>
<td>??</td>
</tr>
</tbody>
</table>

NOP instructions are in red.
Extensions

Encoding Restrictions
Synthesis of Equivalent Snippets
Finding the Shortest Program
Synthesizing Decoders
Input State Preconditions
Integration with Exploit Generation
Input State Preconditions

Conceptually

call $+5
pop ebx
; ebx contains this address
; ... rest of shellcode ...

- It may be impossible to implement something critical within a given encoding.
  - Many shellcodes must locate themselves in memory.
    - GETPC: retrieve the current instruction pointer.
- What happens if we can’t encode GETPC?
Input State Preconditions

Conceptually

- What if, by virtue of our exploit scenario, we know that [esi+4] contains a pointer into the shellcode?
- We can avoid the need for a generic GETPC (that we can’t encode) by synthesizing a shellcode that only works under that assumption.
- This is just an implication based on the input state.

\[ \exists \text{COMPONENTS} \rightarrow \forall \text{INPUTS} \]
\[ \phi \text{Program} \]
\[ \text{INPUTS}[\text{mem}[esi+4]] = \& \text{shellcode} \implies \phi \text{Functionality} \]

- This formula synthesizes a program that is only valid under the assumption that mem[esi+4] == &shellcode.
Extensions

Encoding Restrictions
Synthesis of Equivalent Snippets
Finding the Shortest Program
Synthesizing Decoders
Input State Preconditions

Integration with Exploit Generation
Integration with Exploit Generation

```c
input = recv();
if(!validate(input)) return;
trans = transform(input);
vuln_exec(trans);
```

- Model the execution path from `recv` to `vuln_exec`.
- Synthesize shellcode at the point where `vuln_exec` runs.
- This implicitly models all validation and transformation.
- No need to specify encoding constraints explicitly.
Discussion

Limitations
Evaluation
Future Work
Source Release
Limitations
Constraints Can Be Hard

```plaintext
output = cryptohash(input);
vulnerability(output);
```

- Constraints can be difficult to solve.
- In this example, we need to invert `cryptohash` to generate qualifying shellcode.
  - Second preimage problem.
- Automated exploit generation has the same problem.
Limitations
Can’t Quantify Over Arrays

\[ \exists \text{mc}[256]: \text{char} \ Array \cdot \cdot \cdot \]

- **Problem**: YICES won’t let us quantify over arrays.
Limitations
Can’t Quantify Over Arrays

► Problem: YICES won’t let us quantify over arrays.
► Solution: Quantify over bytes; simulate array access.
  ► Not a very serious limitation for synthesizing arrays.
Limitations
Can’t Quantify Over Arrays

\[ \exists \text{mc}[256]: \text{char Array} \cdot \]
\[ \forall \text{in: state} \cdot \]
\[ \ldots \]

- **Problem**: YICES won’t let us quantify over arrays.
- This means we can’t use arrays as part of our state.
  - No big deal for registers/flags.
  - However, SMT-based analyses use arrays for memory accesses.
    - Therefore, can’t represent memory as part of the state.
    - Therefore, can’t model instructions that manipulate memory.
- An implementation limitation, not a mathematical one.
- A serious limitation, but not necessarily a permanent one.
  - If any solver supports array quantification, we can use it.
  - I didn’t check whether Z3 was suitable.
Limitations

Theoretical Limitations

- Can only synthesize “up to N” instructions.
  - Can’t synthesize “an arbitrarily-long program”.
- Decoder variant/invariant templates are hard-coded.
  - Other iteration orders are possible.
  - We leave generalization to future work.
## Limitations

### Current Capabilities

<table>
<thead>
<tr>
<th>Capability</th>
<th>SIMPLEMC</th>
<th>X86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Straight-line, No Memory</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Straight-line, Memory</td>
<td>XS</td>
<td>XS</td>
</tr>
<tr>
<td>Equivalent Snippets</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shortest Solution</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decoder Synthesis</td>
<td>✓</td>
<td>XS</td>
</tr>
<tr>
<td>Input Preconditions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Exploit Generation</td>
<td>XP</td>
<td>XE</td>
</tr>
</tbody>
</table>

**Legend:**
- ✓: support is present.
- XS: not supported due to solver limitations.
- XE: not supported due to external requirements.
- XP: not supported due to pointlessness.
## Evaluation

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMPLEMC 2-line $r0 = 0$</td>
<td>0.0</td>
</tr>
<tr>
<td>SIMPLEMC shortest 2-line $r0 = 0$</td>
<td>6</td>
</tr>
<tr>
<td>SIMPLEMC longest 2-line $r0 = 0$</td>
<td>0.0</td>
</tr>
<tr>
<td>SIMPLEMC empty decoder</td>
<td>127</td>
</tr>
<tr>
<td>SIMPLEMC exclude bytes decoder</td>
<td>153</td>
</tr>
<tr>
<td>X86 3-line $eax = 12345678h$</td>
<td>0.6</td>
</tr>
</tbody>
</table>
Future Work

- Stochastic superoptimization
  - Perhaps a generative model for mutations?
- Specialize YICES EXISTS/FORALL solver
  - Profile for obvious bottlenecks with the general solver
  - Custom implicant generalization heuristic?
  - Custom SMT theory for X86?
Source release includes many YICES scripts that demonstrate the features shown in this presentation.

- Both X86 and the SIMPLEMC language.

Soon I’ll announce the URL on my twitter account, @RolfRolles.
New Course Offering
Support Weird Computer Security Research

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.
  - Dozens of exercises using SMT solvers.
  - Exercises applying SMT to binary analysis.
  - Code an SMT solver, X86 $\mapsto$ IR translator, ROP compiler$^1$.

- Available now for private offerings!
- See website for public classes (January, Maryland, USA).

$^1$ROP compiler application subject to potential replacement pending forthcoming regulation of the computer security industry.
Conclusion

Any Questions?

- This work broke the ground on automated shellcode synthesis with arbitrary encoding restrictions.
- Works decently for small sequences with simple restrictions.
- More work is necessary for scalability and memory operations.
- YICES source code is available for further experimentation.