

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.

Overview

Classical Memory-Corruption Exploitation

Synesthesia

- 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.

Greetings everybody, and thanks for coming to my talk. The idea is about automatically creating machine code programs in the situation where the code must obey encoding restrictions. This is an idea that popped into my head about four years ago. I was largely too busy to act on it except for jotting down some thoughts in a notebook from time to time. In the meantime, YICES implemented a solver for the types of equations I needed to solve. I finally got the chance to give it a try recently. The problem is basically solved mathematically, but the current implementation suffers from some limitations of modern-day SMT solvers.

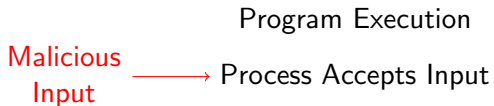
Synesthesia

Classical Memory-Corruption Exploitation

Program Execution

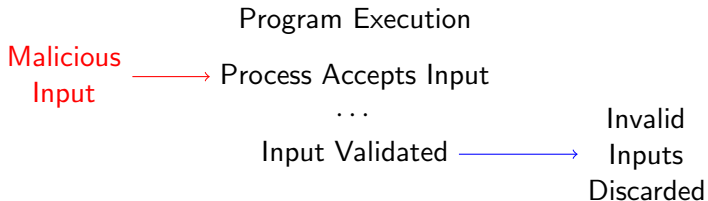
Synesthesia

Classical Memory-Corruption Exploitation



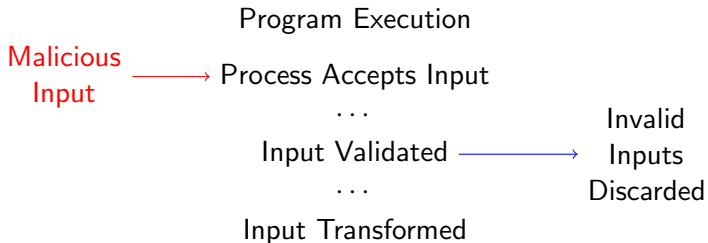
Synesthesia

Classical Memory-Corruption Exploitation



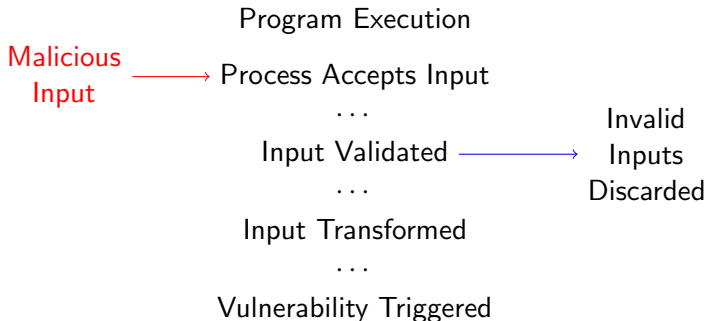
Synesthesia

Classical Memory-Corruption Exploitation



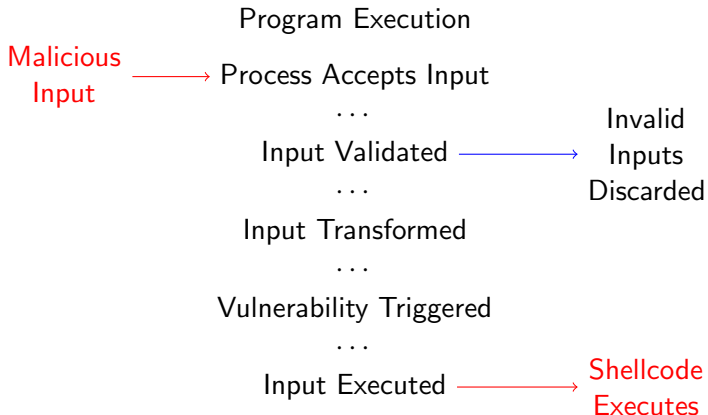
Synesthesia

Classical Memory-Corruption Exploitation



Synesthesia

Classical Memory-Corruption Exploitation



Synesthesia: A Modern Approach to Shellcode Generation

Overview

Classical Memory-Corruption Exploitation

Synesthesia



To briefly review the context of this research, this is an outline of our classical memory corruption exploitation scenario. First, the program accepts input from some outside source. Next, the program may validate the input somehow, for example, ensuring that its input is alphanumeric. As the program executes, the input may be transformed from its original representation. At some point, the input triggers the execution of a vulnerability, and finally, the input is treated as machine code and executed. Of course, exploit mitigations like NX may complicate the situation.

Synesthesia

Restrictions on the Shellcode

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

Restrictions are arbitrary and vary per vulnerability.

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Restrictions on the Shellcode

Synesthesia

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

Restrictions are arbitrary and vary per vulnerability.

The program may place restrictions on the input. For example, it may verify that the input consists of printable characters only, or alphanumeric characters only, or really, any arbitrary restriction. Some common ones are listed on this slide.

Synesthesia

Restrictions on the Shellcode

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES	POP ES
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS	POP SS
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	rAX	rCX	rDX	rBX	rSP	rBP	rSI	rDI
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc, Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv

- ▶ 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.

Synesthesia

Practical Effects of Shellcode Restrictions

Various Ways to Set `eax` to `0h`

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

No NULL bytes

Synesthesia

Practical Effects of Shellcode Restrictions

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

No '%' (25) bytes

Synesthesia

Practical Effects of Shellcode Restrictions

Various Ways to Set `eax` to `0h`

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

All bytes are printable

Synesthesia

Practical Effects of Shellcode Restrictions

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

Synesthesia

Practical Effects of Shellcode Restrictions

Various Ways to Set `eax` to `0h`

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

All bytes are alphanumeric

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Accomplishing Tasks Within Shellcode Restrictions

Synesthesia

Various Ways to Set `eax` to `0x`

```
BB 00 00 00 00  mov  eax, 0
33 00          xor  eax, eax
F8            cld
1E 00          abb  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        xrb  al, 30h
```

All bytes are alphanumeric

Given that shellcode is ultimately just a program, it has to do similar things to any machine code program. It has to set registers to values, read and write to memory, invoke API functions, etc. And given that we are talking about input restrictions, in order to accomplish those tasks, we need to do those things using only machine code instructions that fit within the input restriction. This slide shows examples of different ways to set the `eax` register to `0`, under a variety of encoding restrictions. Some examples are suitable for a given restriction, and some are not. So if we were precluded from using NULL bytes, we could use any of the highlighted sequences.

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Accomplishing Tasks Within Shellcode Restrictions

Synesthesia

Synesthesia

Practical Effects of Shellcode Restrictions

Various Ways to Set `eax` to `0x`

```
BB 00 00 00 00  mov  eax, 0
33 00          xor  eax, eax
F0            cld
1E 00          add  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 are alphanumeric

And if we were prohibited from using percentage characters, we could use the highlighted solutions.

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Accomplishing Tasks Within Shellcode Restrictions

Synesthesia

Synesthesia

Practical Effects of Shellcode Restrictions

Various Ways to Set `eax` to `0x`

```
BB 00 00 00 00  mov  eax, 0
33 00          xor  eax, eax
F0            cld
1E 00          abb  eax, eax
25 56 34 42 24  and  eax, 24423456h
25 28 48 21 42  and  eax, 42214828h
FA 30          push 30h
58            pop  eax
34 30          xor  al, 30h
```

All bytes are alphanumeric

And if all bytes had to be printable, we could use these.

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Accomplishing Tasks Within Shellcode Restrictions

Synesthesia

Synesthesia

Practical Effects of Shellcode Restrictions

Various Ways to Set `eax` to `0x`

```
BB 00 00 00 00  mov  eax, 0
33 00           xor  eax, eax
F0             cli
1E 00         add  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 are alphanumeric

And if all bytes must be ASCII, we could use these.

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Accomplishing Tasks Within Shellcode Restrictions

Synesthesia

Synesthesia

Practical Effects of Shellcode Restrictions

Various Ways to Set `eax` to `0x`

```
BB 00 00 00 00  mov  eax, 0
33 00          xor  eax, eax
F0            cli
1E 00         jmp  eax, eax
25 56 34 42 24  and  eax, 24423456h
25 28 48 21 42  and  eax, 42214828h
FA 30         push 30h
58           pop  eax
34 30         xrb  al, 30h
```

All bytes are alphanumeric

And if all bytes had to be alphanumeric, we could use these. So in general, any given snippet may be useful under some encoding restrictions, and useless under others.

Overview

Existing Solution: Shellcode Encoders

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

- ▶ Begin with unencoded shellcode.

Overview

Existing Solution: Shellcode Encoders

50	48	45	45	4B	48
4A	50	4D	47	46	4F
45	44	4B	4E	4C	4B
50	48	44	50	45	4C
4F	50	47	48	42	42
4F	50	47	48	42	42

- ▶ Begin with unencoded shellcode.
- ▶ Produce encoded shellcode (within encoding restriction).

Overview

Existing Solution: Shellcode Encoders

```
50 48 45 45 4B 48
4A 50 4D 47 46 4F
45 44 4B 4E 4C 4B
50 48 44 50 45 4C
4F 50 47 48 42 42
4F 50 47 48 42 42
```

```
53 push ebx
51 push ecx
53 push ebx
55 push ebp
53 push ebx
46 inc esi
56 push esi
44 inc esp
; ...
```

- ▶ Begin with unencoded shellcode.
- ▶ Produce encoded shellcode (within encoding restriction).
- ▶ Produce decoder (within encoding restriction).

Overview

Shellcode Encoders

Overview

Overview

Existing Solution: Shellcode Encoders

```
50 48 45 45 48 48
44 50 42 47 46 4F
45 44 48 4E 4C 48
50 48 44 50 45 4C
4F 50 47 48 42 42
4F 50 47 48 42 42
```

```
53 push ebx
51 push ecx
53 push ebx
55 push ebp
53 push ebx
48 inc esi
56 push esi
44 inc esp
; ...
```

- Begin with unencoded shellcode.
- Produce encoded shellcode (within encoding restriction).
- Produce decoder (within encoding restriction).

One of the major ideas in this area is that, given that it is onerous to write an entire shellcode within a restriction, that we can take an existing shellcode, encode it to lie within a given restriction, and then create a decoder whose machine code lies within the restriction. This way, we reduce the amount of code that we need to write within the 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: A Modern Approach to Shellcode Generation

Overview

Shellcode Encoders

Overview

Overview

Pros and Cons of Shellcode Encoders

Pros:

- It often works
- Can handle common cases automatically

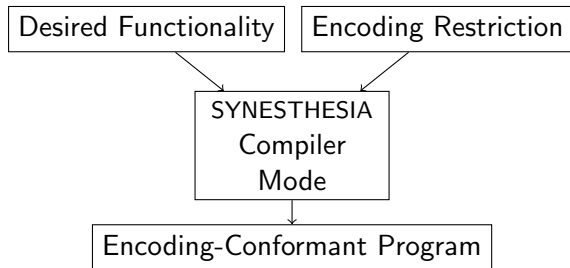
Cons:

- 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

Encoders often work, and they can handle some of the common cases automatically. But they operate at the cost of expanding the shellcode size and potentially introducing pattern sequences that can be detected by IDS or HIPS products. Also, they require manual analysis of the instruction set to produce an encoder/decoder generator, they are not guaranteed to work, and the code is usually very ugly.

Overview

Synesthesia: Compiler Mode



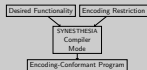
- ▶ Synesthesia can act like a compiler that also inputs an encoding restriction.

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Synesthesia

Overview

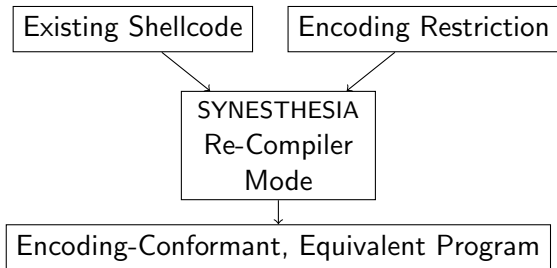


- Synesthesia can act like a compiler that also inputs an encoding restriction.

This work introduces Synesthesia, my take on the encoding-restriction problem. Synesthesia can work in several different ways. One of them is like a compiler, where you instruct Synesthesia what the desired code needs to do, and also provide it with a description of legal encodings for the shellcode.

Overview

Synesthesia: Re-Compiler Mode



- ▶ Synesthesia can input an existing code fragment, and find an equivalent version that also satisfies an encoding restriction.

Synesthesia: A Modern Approach to Shellcode Generation

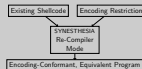
Overview

Synesthesia

Overview

Overview

Synesthesia: Re-Compiler Mode

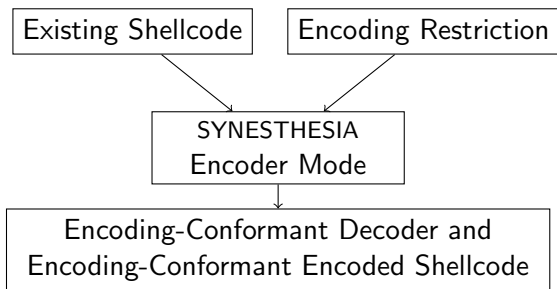


- Synesthesia can input an existing code fragment, and find an equivalent version that also satisfies an encoding restriction.

Another thing that Synesthesia can do is take some existing shellcode fragment, and find an equivalent sequence for it that lies within the legal encodings.

Overview

Synesthesia: Encoder Mode



- ▶ Synesthesia can take existing binary shellcode blobs, and automatically encode them (and generate a decoder) to lie within the specified encoding restriction.

Synesthesia: A Modern Approach to Shellcode Generation

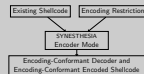
Overview

Synesthesia

Overview

Overview

Synesthesia: Encoder Mode



- Synesthesia can take existing binary shellcode blobs, and automatically encode them (and generate a decoder) to fit within the specified encoding restriction.

More experimentally, Synesthesia can try to encode an existing shellcode automatically, as well as generate a decoder for it.

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

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Synesthesia

Overview

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
 - Encouraging 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

Synesthesia has some nice properties that are unique to tools in this category. It is fully automated and does not require any manual analysis to determine how to perform a given operation within a given restriction. It is also based on static analysis, meaning that it does not run instructions on the processor. It is flexible: it can try to find solutions to any encoding restriction that can be specified as a first-order predicate, and the idea is not specific to any given processor. It is exhaustive: it searches a space of all legal programs to find a solution among all modelled instructions, and it can even find all possible solutions within a given restriction. It is optimal; you can find the shortest solution within a given restriction. It is metamorphic: it does not use pre-generated patterns, and it may produce a different output every time. It can potentially produce self-modifying code, although that idea is not explored further in this presentation.

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.

Synesthesia: A Modern Approach to Shellcode Generation

Overview

Synesthesia

Overview

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.

It's important to note that Synesthesia is research. It works well for short, simple sequences, but it can exhibit long runtimes for more complicated sequences. Also, there are some practical limitations, some of which are the standard ones for anything relying on an SMT solver, and some of which are specific to my particular research implementation.

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 \sim (**not**) or $-$ (**neg**)?

We began by **enumerating all possible programs**:

$y = \sim x;$	$y = -x;$
$z = \sim y;$	$z = \sim y;$
$y = \sim x;$	$y = -x;$
$z = -y;$	$z = -y;$

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

- Synthesizing C-Like Programs

- Symbolic Program Synthesis

Question: is it possible to create the function `++i` by using:

- Two statements, where:
- Each statement has one operator, and:
- Each operator is `^` (`not`) or `-` (`neg`)?

We began by enumerating all possible programs:

<code>y = ^x;</code>	<code>y = -x;</code>
<code>z = ^y;</code>	<code>z = ^y;</code>
<code>y = ^x;</code>	<code>y = -x;</code>
<code>z = -y;</code>	<code>z = -y;</code>

Synesthesia is based on symbolic program synthesis, so we'll give a basic walkthrough of that using an example I've given before. We'll show how to adapt this idea from C-like programs into assembly programs.

So here's our example. Let's say I want to synthesize a program that increments a 32-bit integer, that the program is two lines long, and that each of the lines contains a single application of either the `not` or `neg` operation. For clarity, the slide lists all possible programs matching this description.

Symbolic Program Synthesis

Step #2: Encapsulate Variation into Components

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

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

$$\overrightarrow{\text{COMPONENTS}} = \langle \text{bool } \text{bop1}, \text{bool } \text{bop2} \rangle$$

- ▶ `bool bop1`: is first operation `~`, or `-`?
- ▶ `bool bop2`: is second operation `~`, or `-`?

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

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

y = x ;	y = x ;
z = y ;	z = y ;
y = x ;	y = x ;
z = y ;	z = y ;

COMPONENTS = (bool bop1, bool bop2)

- bool bop1: is first operation "+", or "-"
- bool bop2: is second operation "+", or "-"

We note that each line performs one of two operations, **not** or **neg**. Since there are only two possibilities, we can use a **bool** value to represent which operation is performed on each line. These values are called the **components** of our symbolic program.

Symbolic Program Synthesis

Step #3: Create Symbolic Representation in terms of Components

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

```
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**: `int x`.

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

Question is now: can we set `bop1` and `bop2` so that `f(x) == x+1` for all `x`?

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

 - Synthesizing C-Like Programs

 - Symbolic Program Synthesis

Symbolic Program Synthesis

Step 2.3: Create Symbolic Representation in terms of Components

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

```
bool bsp1; ◀  
bool bsp2; ◀  
  
int f(int a)  
{  
  int y = bsp1 ? -a : "a"; ◀  
  int x = bsp2 ? -y : "y"; ◀  
  return x;  
}
```

Function `f` takes one input: `int a`.

`INPUTS = (int a)`

Question is now: can we set `bsp1` and `bsp2` so that `f(x) == -x` for all `x`?

Now, we can write a bit of C code that represents all of our possible programs, and which behaves like any of the candidate programs based upon the value of the components. This here is the main trick behind symbolic program synthesis. We represent all possible programs and use data items to specify a single one. Now, the question becomes: is there a way to set the data items such that the symbolic program has the behavior that we desire?

Symbolic Program Synthesis

Step #4: Create Synthesis Formula

- ▶ We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bop1</code> , <code>bop2</code>	

Symbolic Program Synthesis

Step #4: Create Synthesis Formula

- ▶ We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bob1</code> , <code>bob2</code>	$\exists \text{ bob1, bob2} \in \mathbf{Bool} \cdot$

Symbolic Program Synthesis

Step #4: Create Synthesis Formula

- ▶ We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bob1</code> , <code>bob2</code>	$\exists \text{ bob1, bob2} \in \mathbf{Bool} \cdot$
Such that, for all values of <code>x</code>	

Symbolic Program Synthesis

Step #4: Create Synthesis Formula

- ▶ We need to rephrase the question mathematically:

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>	$\forall x \in \mathbf{BV}[32] \cdot$

Symbolic Program Synthesis

Step #4: Create Synthesis Formula

- ▶ We need to rephrase the question mathematically:

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>	$\forall x \in \mathbf{BV}[32] \cdot$
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

Symbolic Program Synthesis

Step #4: Create Synthesis Formula

- ▶ We need to rephrase the question mathematically:

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>	$\forall x \in \mathbf{BV}[32] \cdot$
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 <code>true</code> ?	<code>z == x+1</code>

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

- Synthesizing C-Like Programs

- Symbolic Program Synthesis

- We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bug1, bug2</code> s.t. <code>bug1, bug2 ∈ Bool</code>	$\exists \text{ bug1, bug2 } \in \text{Bool}$
Such that, for all values of <code>a</code>	$\forall a \in \text{BV}[32]$
In the code	
<code>y = bug1 ? -a : *a;</code>	<code>let y = bug1 ? -a : *a in</code>
<code>a = bug2 ? -y : *y;</code>	<code>let a = bug2 ? -y : *y in</code>
<code>a == *a</code> is always true?	<code>a == *a</code>

Next we need to phrase the question mathematically. In English, our question is: can we find values for the components

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

- We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bop1</code> , <code>bop2</code> s.t. <code>bop1</code> , <code>bop2</code> C <code>Bool</code>	$\exists \text{ bop1, bop2} \in \text{Bool}$
Such that, for all values of <code>a</code>	$\forall a \in \text{BV}[32]$
In the code	
<code>y = bop1 ? -a : *a;</code>	$\text{let } y = \text{bop1 ? -a : *a in}$
<code>a = bop2 ? -y : *y;</code>	$\text{let } a = \text{bop2 ? -y : *y in}$
<code>a == *a</code> is always true?	$a == *a$

The mathematical reification of this part of the question uses something called an **existential quantifier**, the backwards “E”, pronounced **there exists**. **There exists** values of `bop1` and `bop2`, both `bool` values

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

- We need to rephrase the question mathematically:

English	Mathematics
Are there values of $loop1, loop2$	$\exists loop1, loop2 \in \mathbf{Bool}$
Such that, for all values of x	$\forall x \in \mathbf{BV}[32]$
In the code	
$y = loop1 ? x : 0;$	let $y = loop1 ? x : 0;$ in
$x = loop2 ? y : 0;$	let $x = loop2 ? y : 0;$ in
$x == 0$ is always true?	$x == 0$

Now we specify the behavior of the program. Our choices for the components need to work for all values of the 32-bit integer x .

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

- We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bug1, bug2</code> ?	$\exists \text{ bug1, bug2} \in \text{Bool}$
Such that, for all values of <code>x</code>	$\forall x \in \text{BV}[32]$
In the code	
<code>y = bug1 ? -x : *x;</code>	<code>let y = bug1 ? -x : *x in</code>
<code>x = bug2 ? -y : *y;</code>	<code>let x = bug2 ? -y : *y in</code>
<code>x == x+1</code> is always true?	<code>x == x+1</code>

To represent this mathematically, we use something called a **universal quantifier**, the upside-down “A”, pronounced **for all**. So, **for all** values of the 32-bit integer `x` ...

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

- We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bug1, bug2</code> ?	$\exists \text{ bug1, bug2} \in \text{Bool}$
Such that, for all values of <code>a</code>	$\forall a \in \text{BV}[32]$
In the code	
<code>y = bug1 ? -a : *x;</code>	<code>let y = bug1 ? -a : *x in</code>
<code>a = bug2 ? -y : *z;</code>	<code>let a = bug2 ? -y : *z in</code>
<code>z == 0x1</code> is always true?	<code>a == 0x1</code>

Thanks to SMT solvers, we have a language to describe the ordinary operations used within programs. We can pretty much just translate it line-for-line from C into SMT.

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

- We need to rephrase the question mathematically:

English	Mathematics
Are there values of <code>bug1, bug2</code>	$\exists \text{ bug1, bug2} \in \text{Bool}$
Such that, for all values of <code>x</code>	$\forall x \in \text{BV}[32]$
In the code	
<code>y = bug1 ? -x : *x;</code>	<code>let y = bug1 ? -x : *x in</code>
<code>x = bug2 ? -y : *y;</code>	<code>let x = bug2 ? -y : *y in</code>
<code>z == x+1</code> is always true?	<code>z == x+1</code>

Finally, we specify the desired behavior of our symbolic program. We want the output, the variable `z`, to be the incremented version of the input `x`. And that's our entire formula.

Symbolic Program Synthesis

Step #4: Create Synthesis Formula

Create a synthesis formula consisting of four elements.

Symbol	Description	Contents
$\exists \text{COMPONENTS}$	Exists components	$\exists \text{ bop1, bop2} \in \mathbf{Bool} \cdot$
$\forall \text{INPUTS}$	For all inputs	$\forall \mathbf{x} \in \mathbf{BV}[32] \cdot$
ϕ_{Program}	Program constraint	let $y = \text{bop1} ? -x : \sim x$ in let $z = \text{bop2} ? -y : \sim y$ in
$\phi_{\text{Functionality}}$	Functionality constraint	$z == x+1$

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

- Synthesizing C-Like Programs

- Symbolic Program Synthesis

Create a synthesis formula consisting of four elements.

Symbol	Description	Contents
\exists COMPONENTS	Exists components	$\exists \text{ heap}, \text{ heap} \in \text{Bool}$
\forall INPUTS	For all inputs	$\forall x \in \text{BV}[32]$
ϕ Program	Program constraint	$\text{let } y = \text{heap} ? \rightarrow : 'x \text{ in}$ $\text{let } z = \text{heap} ? \rightarrow : 'y \text{ in}$
ϕ Functionality	Functionality constraint	$z == x+1$

To recap, we had four elements to our synthesis formula. We are looking for values for our components – exists components – such that, for all inputs – for all inputs – the symbolic program (represented by the third term) has the behavior that we desire (represented by the final term).

Symbolic Program Synthesis

Step #5: Solve Synthesis Formula

Solve the synthesis formula with an SMT solver.

$$\text{SMT} \left(\frac{\frac{\exists \text{ bop1, bop2} \in \mathbf{Bool} \cdot}{\forall x \in \mathbf{BV}[32] \cdot}}{\text{let } y = \text{bop1} ? -x : \sim x \text{ in}}}{\text{let } z = \text{bop2} ? -y : \sim y \text{ in}} \right) =$$
$$z == x+1$$

<code>bop1</code>	\mapsto	<code>false</code>	<code>bop2</code>	\mapsto	<code>true</code>
-------------------	-----------	--------------------	-------------------	-----------	-------------------

Solution for $\overrightarrow{\text{COMPONENTS}}$

If the formula is unsolvable, the solver returns UNSAT.

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

Solve the synthesis formula with an SMT solver.

$$\text{SMT} \left(\begin{array}{l} \exists \text{ bool1, bool2} \in \text{Bool} \\ \forall a \in \text{BV}[32] \\ \text{let } y = \text{bool1} ? -a : ~a \text{ in} \\ \text{let } x = \text{bool2} ? -y : ~y \text{ in} \\ x == !a \end{array} \right) =$$

bool1	==	false	bool2	==	true
-------	----	-------	-------	----	------

Solution for COMPONENTS

If the formula is unsolvable, the solver returns UNSAT.

Now we just feed the formula to an SMT solver and ask for a solution. If it's successful, it's going to give us values for the components that invoke the desired behavior. We can see the solution on the slide here. It may be impossible to cause the desired behavior, in which case the SMT solver will tell us that.

Symbolic Program Synthesis

Step #6: Interpret Synthesis Formula Solution

Plug the solution for $\overrightarrow{\text{COMPONENTS}}$ 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 $\overrightarrow{\text{COMPONENTS}}$

bop1	\mapsto	false	◀
bop2	\mapsto	true	◀

Symbolic Program Synthesis

Step #6: Interpret Synthesis Formula Solution

Plug the solution for $\overline{\text{COMPONENTS}}$ into the symbolic program ...

```
int f(int x)
{
  int y = ~x; ◀
  int z = -y; ◀
  return z;
}
```

Solution for $\overline{\text{COMPONENTS}}$

bop1	\mapsto	false	◀
bop2	\mapsto	true	◀

... to obtain the desired program.

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis
 - Synthesizing C-Like Programs
 - Symbolic Program Synthesis

Plug the solution for COMPONENTS into the symbolic program ...

```
int f(int x)
{
  int y = ~x;
  int z = ~y;
  return x;
}
```

Solution for COMPONENTS

tmp1 == false
tmp2 == true

... to obtain the desired program.

Once we have a solution for the components, we just plug them into our symbolic program ...

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing C-Like Programs

Symbolic Program Synthesis

Plug the solution for COMPONENTS into the symbolic program ...

```
int f(int x)
{
  int y = ~x;
  int z = ~y;
  return x;
}
```

Solution for COMPONENTS

tmp1 == false
tmp2 == true

... to obtain the desired program.

And then we get the program that causes our desired behavior.

Symbolic Program Synthesis

More on Synthesis Formulas

Each formula in this presentation has roughly the same structure.

Symbol	Description
$\exists \overline{\text{COMPONENTS}}$	Exists components
$\forall \overline{\text{INPUTS}}$	For all inputs
ϕ_{Program}	Program constraint
$\phi_{\text{Functionality}}$	Functionality constraint

This formula structure has several names in the literature:

- ▶ **Exists/forall**
- ▶ **One quantifier alternation**
- ▶ **Effectively propositional**
- ▶ **Bernays-Schönfinkel**

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

Each formula in this presentation has roughly the same structure.

Symbol	Description
\exists COMPONENTS	Exists components
\forall INPUTS	For all inputs
ϕ PROGRAM	Program constraint
ϕ FUNCTIONALITY	Functionality constraint

This formula structure has several names in the literature:

- **Exists/forall**
- **One quantifier alternation**
- **Effectively propositional**
- **Bornays-Schüpfinkel**

A quick note before we move on. If you end up doing any research in this area, you should be aware that this formula structure has a few names in the literature. I've listed them on this slide in order of increasing formality. This slide is also known as, "why is there an umlaut in the title of your presentation?"

Symbolic Program Synthesis

Extending the Framework: More Operator Types

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;  
}
```

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis
 - Synthesizing C-Like Programs
 - Symbolic Program Synthesis

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

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

Before we move on, we'll note two extensions to the basic idea of symbolic program synthesis. First, in our previous example, there were only two possibilities per line, and so we could use a `bool` value to represent them. Of course, if we had more than two operators, we would need more than a single `bool` to represent them. This slide shows an example using three operators, and an 8-bit value to choose between them. Pretty simple.

Symbolic Program Synthesis

Extending the Framework: Unspecified Constants

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

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

Symbolic Program Synthesis

Extending the Framework: Unspecified Constants

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

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

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

$\exists \langle op \in \mathbf{Bool}, c \in \mathbf{BV}[8] \rangle$.

$\forall x \in \mathbf{BV}[8]$.

let $y = op ? x + c : x \wedge c$ in

$y == \sim x$

Symbolic Program Synthesis

Extending the Framework: Unspecified Constants

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

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

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

$\exists \langle op \in \mathbf{Bool}, \triangleright c \in \mathbf{BV}[8] \langle \rangle \cdot$

$\forall x \in \mathbf{BV}[8] \cdot$

let $y = op ? x + c \langle \rangle : x \wedge c \langle \rangle$ in

$y == \sim x$

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

Solution: $op \mapsto \text{false}, \triangleright c \mapsto 0xFF \langle \rangle$.

I.e. $f(x)$ is $x \wedge \triangleright 0xFF \langle \rangle$.

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

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

```

Let's synthesize  $f(x) == 'x$ .
bool op;
char c;
int f(char x)
{
  y = op ?
  x + c :
  x ^ c;
  return y;
}

```

$\exists op \in \text{Bool}, \triangleright c \in \text{BV}[8]$
 $\forall x \in \text{BV}[8]$:
 $\text{let } y = \text{op} ? x + c : x ^ c \text{ in}$
 $y == 'x$

Constants are components, so solutions must include values for the constants.
Solution: $op \mapsto \text{false}, \triangleright c \mapsto \text{0xFF}$
i.e. $f(x)$ is $x ^ \triangleright \text{0xFF}$

Secondly, a slightly more complicated extension to the idea. This example relies upon an integer whose value is not specified. So, depending on the value of `op`, it either ADDs or XORs the input with the value of the constant `c`. `c` is a component, so the SMT solver will have to provide a value for `c` for any given synthesis formula.

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing C-Like Programs
 - └ Symbolic Program Synthesis

We can extend the idea to incorporate unspecified constants:

```

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

```

Let's synthesize $f(x) == 'x$.

$\exists op \in \text{Bool}, c \in \text{BV}[8]$
 $\forall x \in \text{BV}[8]$
 let $y = op ? x + c : x * c$ in
 $y == 'x$

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

Solution: $op \mapsto \text{false}, c \mapsto \text{0xFF}$
 i.e. $f(x)$ is $x * \text{0xFF}$

So, for example, if we wanted to use this template to generate the logical **not** function, here would be our SMT formula. Simple, really; the only thing to note is that **c** is a component in the first line.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis
 Synthesizing C-Like Programs
 Symbolic Program Synthesis

We can extend the idea to incorporate unspecified constants:

```

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

```

Let's synthesize $f(x) == 'x'$.

$\exists op \in \text{Bool}, c \in \text{BV}[8]$
 $\forall x \in \text{BV}[8]$
 $let\ y = op ? x + c : x * c\ in$
 $y == 'x'$

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

Solution: $op \mapsto \text{false}, c \mapsto \text{0xFF}$
 i.e. $f(x)$ is $x * \text{0xFF}$

And then, if we were to solve this, the solution would be operator XOR, constant value **0FFh**. The solution gives a value for the unspecified constant **c**.

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.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

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 state
3. We write a simulator for SIMPLE.
 - A function to update SIMPLE machine state
 - A function to simulate SIMPLE operations
 - A function to simulate SIMPLE instructions
4. We synthesize SIMPLE programs.

Moving on, we're going to adapt the idea to synthesizing assembly language programs. Most of this presentation will use examples in a language I made up called SIMPLE. I have provided an X86 implementation as well, but X86 is too detailed to describe in an hour. See the source code for full details.

So the first thing we're going to do is describe SIMPLE, and write a C implementation of it as a language, and a simulator for it. Then we can just use the techniques from the previous section to synthesize SIMPLE programs in terms of its C representation.

Synthesizing Assembly Programs

Transitioning from Synthesizing C Programs

Synthesizing C Programs

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

Synthesizing ASM Programs

```
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 ◀ become **assembly language instructions**.
- ▶ Inputs and outputs ◀ become **machine states**.
 - ▶ Register and flag values, and/or memory contents.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Transitioning from Synthesizing C Programs

Synthesizing C Programs	Synthesizing ASM Programs
<code>bool bop1; ◀</code>	<code>Instruction i1; ◀</code>
<code>bool bop2; ◀</code>	<code>Instruction i2; ◀</code>
<code>int f(int x) ◀</code>	<code>state *i(state *in) ◀</code>
<code>{</code>	<code>{</code>
<code> int y = bop1 ? -x : *x;</code>	<code> state *s1 = EvaluateOp(in, i1);</code>
<code> int z = bop2 ? -y : *y;</code>	<code> state *s2 = EvaluateOp(s1, i2);</code>
<code> return z; ◀</code>	<code> return s2; ◀</code>
<code>}</code>	<code>}</code>

Differences in synthesizing ASM programs versus C:

- Components ◀ become **assembly language instructions**.
- Inputs and outputs ◀ become **machine states**.
 - Register and flag values, and/or memory contents.

And, just so you have an idea of where we're going with all of this, here's what our synthesis formulas for SIMPLE are going to look like. On the left is the example from the previous section: two `bool` components; the synthesis function takes as input a 32-bit integer `x`, performs two operations, and returns a 32-bit integer value.

On the right, we have our synthesis formula for SIMPLE. It has two components as well, except the components are `Instruction` structures. Its synthesis function takes as input something called a `state`, a machine state. It performs two operations – two instructions – and returns the transformed machine state.

Synthesizing Assembly Programs

SIMPLE Assembly Language

- ▶ SIMPLE has 8 32-bit registers, `r0` to `r7`.
- ▶ Instructions are below; they work like you would expect.
 - ▶ `rX` and `rY` stand for any of the 32-bit registers.
 - ▶ `imm32` stands for any 32-bit constant value.

```
xor rX, rY      add rX, rY      mov rX, rY
inc rX          dec rX          neg rX          not rX
add rX, imm32  xor rX, imm32  and rX, imm32  or rX, imm32
```

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

- SIMPLE has 8 32-bit registers, r0 to r7.
- Instructions are below; they work like you would expect.
 - r1 and r7 stand for any of the 32-bit registers.
 - imm32 stands for any 32-bit constant value.

```
xor rX, rY    add rX, rY    mov rX, rY
inc rX        dec rX      neg rX      not rX
add rX, imm32 xor rX, imm32 and rX, imm32 or rX, imm32
```

SIMPLE is a simple language. It just has eight 32-bit registers, and 11 opcodes. It has binary XOR, ADD, and MOV. It has unary INC, DEC, NEG, and NOT. Finally, it has binary ADD with a constant, XOR, AND, and OR. They work exactly like you'd expect, no tricks up my sleeve.

Synthesizing Assembly Programs

SIMPLE Assembly Language, Symbolic Representation of Opcodes

```
enum Simple {  
    XorRegReg,  
    AddRegReg,  
    MovRegReg,  
    IncReg,  
    DecReg,  
    NegReg,  
    NotReg,  
    AddRegImm,  
    XorRegImm,  
    AndRegImm,  
    OrRegImm,  
};
```

We define an enumeration with one entry per instruction type.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Synthesizing Assembly Programs

SIMPLE Assembly Language, Symbolic Representation of Opcodes

```
enum Simple {  
    EvrOpAgg,  
    AddOpAgg,  
    MovOpAgg,  
    IncOp,  
    DecOp,  
    EngOp,  
    RetOp,  
    AddOpImm,  
    IncOpImm,  
    DecOpImm,  
    CallOpImm,  
};
```

We define an enumeration with one entry per instruction type.

To represent SIMPLE in C, we'll define an enumeration, with one entry for each opcode.

Synthesizing Assembly Programs

SIMPLE Assembly Language, Symbolic Representation of Instructions

```
struct Instruction {
    Simple op;
    int lhsRegNum;
    int rhsRegNum;
    uint32 imm32;
};
```

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).

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

```
struct Instruction {
    .: Single op;
    .: int lhsRegNum;
    .: int rhsRegNum;
    .: uint32 imm32;
};
```

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).

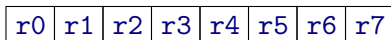
Next, we define an **Instruction** structure. Each instruction has an opcode and a left-hand register number. Some instructions have a right-hand register number, and some instructions have a right-hand 32-bit constant value.

Synthesizing Assembly Programs

SIMPLE Assembly Language, Machine State

We model the machine state as an array. E.g., `state[4]` is `r4`, etc.

```
typedef uint32[8];
```



Machine State

For more complex assembly languages, we'll need (at least) flags and memory.

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

 - Synthesizing Assembly Programs

 - Synthesizing Assembly Programs

We model the machine state as an array. E.g., `state[4]` is `r4`, etc.

```
typedef state_t state[8];
```

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

Machine State

For more complex assembly languages, we'll need (at least) flags and memory.

To model SIMPLE machine states, since it only has 8 registers, we'll just use an array with 8 32-bit integers. For more complex assembly languages, we'll need to model flags and memory also.

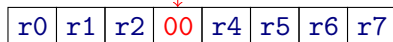
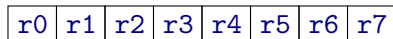
Synthesizing Assembly Programs

SIMPLE Assembly Language, Updating the Machine State

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)`



New Output State (Input State Copied, `r3` Updated)

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Synthesizing Assembly Programs
SIMPLC Assembly Language, Updating the Machine State

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` `r3` `r4` `r5` `r6` `r7`

New Output State (Input State Copied, `r3` Updated)

Next, we write a little function to update a `state`. You give it an existing state, a register number, and a new value for that register. It copies the state and modifies the specified register to the specified value. Note that it creates new `states` rather than modifying existing ones.

Synthesizing Assembly Programs

SIMPLE Assembly Language, Updating the Machine State

```
state *Update(state *state, int regNum, uint32 value)

    state *out = new state;
    memcpy(out, state, sizeof(*state));
    out[regNum] = value;
    return out;
```

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Synthesizing Assembly Programs
SIMPLC Assembly Language, Updating the Machine State

```
state *Update(state *state, int register, uint32 value)
{
    state *out = new state;
    memcpy(out, state, sizeof(*state));
    out[register] = value;
    return out;
}
```

Skip this slide; it's only for completeness.

Synthesizing Assembly Programs

SIMPLE Assembly Language, Emulation

```
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.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

```
uint32 PerformBin(Simple op, uint32 l, uint32 r, uint32 i)
{
    switch(op)
    {
        case XorRegReg: return l ^ r; case AddRegReg: return l + r;
        case AndReg:   return l & i; case SubReg:   return l - i;
        case RorReg:   return l; case RorReg:   return l;
        case AddRegImm: return l + i; case AndRegImm: return l & i;
        case AndRegImm: return l & i; case OrRegImm:  return l | i;
    }
}
```

• Emulating SIMPLE is very easy.

SIMPLE is, well, simple to simulate. This function just performs some specified SIMPLE operation. You give it an opcode, a value for the left-hand and right-hand registers, and a 32-bit value, and it performs the desired operation. For example, if the operation is XOR two registers, it XORs them and returns the value. If the operation is ADD a register and an immediate, it ADDs them and returns the value. Simple.

Synthesizing Assembly Programs

SIMPLE Assembly Language, Emulation

```
state *EmulateOne(state *state, Instruction *i)

    uint32 l = state[i->lhsRegNum];
    uint32 r = state[i->rhsRegNum];
    uint32 v = PerformOne(i->op,l,r,i->imm32);
    return Update(state, i->lhsRegNum, v);
```

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.

Synesthesia: A Modern Approach to Shellcode Generation

└ Symbolic Program Synthesis

└─ Synthesizing Assembly Programs

└─ Synthesizing Assembly Programs

```
state *EmulateOne(state *state, Instruction *i)
uint32 l = state[->instruction];
uint32 r = state[->instruction];
uint32 v = PerformOne(i->op,l,r,i->imm32);
return Update(state, i->instruction, v);
```

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.

And finally, this is the last piece of our simulator. This function, `EmulateOne`, takes in a `state` and an `Instruction`, fetches the operand values from the `state`, calls the function from the previous slide to return the value, and then returns a new `state` that is updated with the results of the instruction.

Synthesizing Assembly Programs

Comparison with Synthesizing C Programs

Synthesizing C Programs

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

Synthesizing ASM Programs

```
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.

Synesthesia: A Modern Approach to Shellcode Generation

└ Symbolic Program Synthesis

└─ Synthesizing Assembly Programs

└─ Synthesizing Assembly Programs

Synthesizing C Programs	Synthesizing ASM Programs
<pre>bool bop1; ◀ bool bop2; ◀ int f(int x) ◀ { int y = bop1 ? -x : x; int z = bop2 ? -y : y; return z; ◀ }</pre>	<pre>Instruction i1; ◀ Instruction i2; ◀ state *f(state *s) ◀ { state *s1 = EvaluateOne(s, i1); state *s2 = EvaluateOne(s1, i2); return s2; ◀ }</pre>

Side is duplicated from before. Now it should make sense.

Now we have everything we need to synthesize SIMPLE programs. This is the same slide from before, but now it should make sense. Our components are two SIMPLE instructions. The synthesis function takes as input a machine **state**. Each line of the synthesis function transforms a **state** based upon the respective SIMPLE instruction, and then it returns the final transformed **state**.

Synthesizing Assembly Programs

Functionality Constraints

To synthesize SIMPLE programs, we need to specify functionality constraints (input/output relationships) in terms of **states**:

$\phi_{\text{Functionality-Increment-r0}}$

```
s2[0] == in[0]+1  &&  
s2[1] == in[1]    &&  
s2[2] == in[2]    &&  
s2[3] == in[3]    &&  
s2[4] == in[4]    &&  
s2[5] == in[5]    &&  
s2[6] == in[6]    &&  
s2[7] == in[7]
```

This constraint specifies: **r0** increments; other registers unchanged.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

To synthesize SIMPLE programs, we need to specify functionality constraints (input/output relationships) in terms of `axtwek`:

```
@FunctionalityIncrement:r0
```

```
ax[0] == in[0]+1 ##  
ax[1] == in[1] ##  
ax[2] == in[2] ##  
ax[3] == in[3] ##  
ax[4] == in[4] ##  
ax[5] == in[5] ##  
ax[6] == in[6] ##  
ax[7] == in[7] ##
```

This constraint specifies: `r0` increments, other registers unchanged.

To synthesize programs, we're going to need to specify the desired functionality in terms of SIMPLE machine states. This slide shows an example where we want the first register `r0` to be incremented, and all other registers to be preserved.

Synthesizing Assembly Programs

All Together

Our synthesis formula is:

$\exists \langle i1 \in \mathbf{Instruction}, i2 \in \mathbf{Instruction} \rangle \cdot$

$\forall in \in \mathbf{State} \cdot$

let $s1 = \mathbf{EmulateOne}(in, i1)$ in

let $s2 = \mathbf{EmulateOne}(s1, i2)$ in

$\phi_{\mathbf{Functionality-Increment-r0}}$

Synthesizing Assembly Programs

All Together

Our synthesis formula is:

$$\exists \langle i1 \in \mathbf{Instruction}, i2 \in \mathbf{Instruction} \rangle \cdot$$
$$\forall in \in \mathbf{State} \cdot$$

let $s1 = \text{EmulateOne}(in, i1)$ in

let $s2 = \text{EmulateOne}(s1, i2)$ in

$$\phi_{\text{Functionality-Increment-r0}}$$

Solution:

$$i1 \mapsto \{\text{AddRegImm}, 0, 0, 1\}$$
$$i2 \mapsto \{\text{OrRegImm}, 0, 0, 0\}$$

i.e.:

```
add r0, 1
```

```
or r0, 0
```

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

- Synthesizing Assembly Programs

- Synthesizing Assembly Programs

```

Our synthesis formula is:
 $\exists i1 \in \text{Instruction}, i2 \in \text{Instruction}.$ 
 $\forall s \in \text{State}.$ 
let s1 = ExecuteState(s, i1) in
let s2 = ExecuteState(s1, i2) in
 $\langle \text{Practicality between } s0$ 

Solution:
i1  $\mapsto$  {ADDReg32, 0, 0, 1}
i2  $\mapsto$  {ORReg32, 0, 0, 0}

I.e.:
add r0, 1
or r0, 0
  
```

Now, a complete synthesis example. Our formula has two instructions as components, takes a machine state as input, performs two operations, and then specifies the desired behavior – the increment behavior described previously.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

```

Our synthesis formula is:
 $\exists (i1 \in \text{Instruction}, i2 \in \text{Instruction})$ 
 $\forall s \in \text{State}$ 
let s1 = ExecuteState(s, i1) in
let s2 = ExecuteState(s1, i2) in
 $\langle \text{Practicality between } s0$ 

Solution:
i1  $\mapsto$  (ADDRegImm, 0, 0, i3)
i2  $\mapsto$  (NOPRegImm, 0, 0, 0)

I.e.:
add r0, 1
or r0, 0

```

So if we solve this, we'll get a solution in terms of the component instructions. By interpreting the `Instruction` structures as actual instructions, we get the solution we see at bottom right. And it makes sense; that program clearly increments `r0` on the first line, and the second line is just a `nop`.

Synthesizing Assembly Programs

Obtaining Alternative Solutions

Let's say we want a solution different from:

```
add r0, 1  
or r0, 0
```

Existing synthesis formula:

```
 $\exists \langle i1 \in \mathbf{Instruction}, i2 \in \mathbf{Instruction} \rangle \cdot$   
 $\forall in \in \mathbf{State} \cdot$   
let  $s1 = \mathbf{EmulateOne}(in, i1)$  in  
let  $s2 = \mathbf{EmulateOne}(s1, i2)$  in  
 $\phi_{\mathbf{Functionality-Increment-r0}}$ 
```

Synthesizing Assembly Programs

Obtaining Alternative Solutions

Let's say we want a solution different from:

```
add r0, 1
or r0, 0
```

Existing synthesis formula:

$\exists \langle i1 \in \mathbf{Instruction}, i2 \in \mathbf{Instruction} \rangle \cdot$

$\forall in \in \mathbf{State} \cdot$

let $s1 = \mathbf{EmulateOne}(in, i1)$ in

let $s2 = \mathbf{EmulateOne}(s1, i2)$ in

$\phi_{\mathbf{Functionality-Increment-r0}}$

Add these new terms:

$i1.op \neq \mathbf{AddRegImm} \ || \ i1.lhsRegNum \neq 0 \ || \ i1.imm32 \neq 1 \ ||$

$i2.op \neq \mathbf{OrRegImm} \ || \ i2.lhsRegNum \neq 0 \ || \ i2.imm32 \neq 0$

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Let's say we want a solution different from:

```
add r0, 1
or r0, 0
```

Existing synthesis formula:

```
3(x) ∈ Instruction, 2 ∈ Instruction
```

```
Via ∈ Slate
```

```
let x1 = MemState(x, 1) in
```

```
let x2 = MemState(x1, 1) in
```

```
2(x) ∈ Instruction
```

```
-----  
Add these new terms:
```

```
12.op 14 AddRegImm || 12.opRegImm 14 0 || 12.imm32 14 1 ||
```

```
12.op 14 SetRegImm || 12.immRegImm 14 0 || 12.imm32 14 0
```

Two more tricks before we move on to synthesizing machine code programs. First, let's say we wanted to find a different solution from the one we were just given. This slide shows the same synthesis formula for the last slide.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Let's say we want a solution different from:

```
add r0, 1
or r0, 0
```

Existing synthesis formula:

```
3(x) ∈ Instruction, 32 ∈ Instruction)
```

Via ∈ Slate:

```
let x1 = Symbolize(x, 13) in
```

```
let x2 = Symbolize(x1, 13) in
```

```
2(x) ∈ Instruction, 32 ∈ Instruction)
```

Add these new terms:

```
12.op |> AddRegImm || 12.immRegImm |> 0 || 12.imm32 |> 1 ||
```

```
12.op |> SetRegImm || 12.immRegImm |> 0 || 12.imm32 |> 0
```

Let's just add some new terms onto the formula. We say, not only must the solution satisfy the functionality constraint, but also, one of the instructions must be different. So either the first opcode is not “add reg, immediate”, or the left-hand register is not `r0`, or the first immediate is not `1`, and so on. So if we solve this formula, we'll get a different solution than the one just given.

Synthesizing Assembly Programs

First Eight Solutions for $r0 = r0 + 1$

The first 8 solutions with 2 instructions for $r0 = r0 + 1$:

<code>add r0, 1</code>	<code>mov r2, r2</code>
<code>or r0, 0</code>	<code>inc r0</code>
<code>dec r0</code>	<code>xor r0, 0</code>
<code>add r0, 2</code>	<code>add r0, 1</code>
<code>not r0</code>	<code>mov r0, r0</code>
<code>neg r0</code>	<code>inc r0</code>
<code>inc r0</code>	<code>add r0, 20D910C5h</code>
<code>mov r0, r0</code>	<code>add r0, 0DF26EF3Ch</code>

Some solutions have `NOP` instructions in them.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

The first 8 solutions with 2 instructions for $r0 = r0 + 1$:

add r0, 1	mov r2, r2
or r0, 0	inc r0
dec r0, 2	mov r0, 0
add r0, 2	add r0, 1
not r0	mov r0, r0
neg r0	inc r0
inc r0	add r0, 000910C5h
mov r0, r0	add r0, 00F26F3Ch

Some solutions have **NOP** instructions in them.

And indeed, here are the first 8 solutions that we get. Many of them contain NOP instructions in red. We also see: subtract one, then add two; the same not/neg trick from before; and add two values that add up to 1.

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`.

Synthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Synthesizing Assembly Programs

Variable-Length Programs

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

state ← {state *i0} {
  state *i1 = EvalState(i1, i1);
  state *i2 = EvalState(i2, i2);
  state *i3 = EvalState(i3, i3);
  state *i4 = EvalState(i4, i4);
  state *i5 = EvalState(i5, i5);
}

return numInstrs == 1 ? a1 :
00000000 numInstrs == 2 ? a2 :
00000000 numInstrs == 3 ? a3 :
00000000 numInstrs == 4 ? a4 :
00000000 numInstrs == 5;
}
```

- 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`.

And, one more extension before moving on. All of our synthesis formulas so far used a fixed number of instructions. We can easily extend the idea to support a variable number of instructions. This example shows up to five instructions, where the number of instructions in the solution is controlled by a variable called `numInstrs`. This construction is more flexible than what's shown previously. We can add extra constraints to fix the number of instructions, like what we were doing previously, or we can specify a range of lengths.

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.

Synesthesia: A Modern Approach to Shellcode Generation

- └ Symbolic Program Synthesis
 - └ Synthesizing Machine-Code Programs
 - └ Synthesizing Machine-Code Programs

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 `Disassembler` 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`.

Now let's move on to synthesizing machine code programs. First, we're going to define a machine code encoding for `SIMPLE`. Next, we'll write a disassembler for `SIMPLE` instructions. And then, we can just use the same ideas as before to synthesize machine code programs.

Synthesizing Machine-Code Programs

Transitioning from 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;
```

```
}
```

Synthesizing SIMPLMC Programs

```
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 ◀.

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

- Synthesizing Machine-Code Programs

- Synthesizing Machine-Code Programs

Synthesizing Machine-Code Programs
Transitioning from Synthesizing ASM Programs

Synthesizing SIMPLE Programs	Synthesizing SIMPLMC Programs
<pre> Instruction i1, i2; state *f(state *in) { state *s1=EmulateOne(in,i1); state *s2=EmulateOne(s1,i2); return s2; } </pre>	<pre> char mc[256]; state *f(state *in) { int i1, i2; Instruction i1, i2; Decode(mc, 0, &i1, &i2); Decode(mc, i1, &i12, &i12); state *s1=EmulateOne(in,i1); state *s2=EmulateOne(s1,i2); return s2; } </pre>

Differences in synthesizing ASM programs versus machine code:

- Components become **machine code bytes**.
- Machine-code formula must **decode instructions**.

As before, here's what our machine-code synthesis formulas are going to look like. For SIMPLE, we just had two components, which were **Instruction** structures. The synthesis function took as input a machine **state**, performed two instructions, and then returned the transformed **state**.

For machine code, our components are an array of machine code bytes. Inside of the synthesis function, the first four lines decode two instructions from the machine code array. The rest of the synthesis function is identical; it takes a state as input, performs two instruction operations, and returns the final transformed state.

Synthesizing Machine-Code Programs

SIMPLE Machine Language

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

<p style="text-align: center;">01 Instruction 00</p> <p style="text-align: center;">00 000 001 Opcode Reg/Op Reg</p>	01 xor r0, r1
<p style="text-align: center;">53 Instruction 00</p> <p style="text-align: center;">01 010 011 Opcode Reg/Op Reg</p>	53 add r2, r3
<p style="text-align: center;">A5 Instruction 00</p> <p style="text-align: center;">10 100 101 Opcode Reg/Op Reg</p>	A5 mov r4, r5

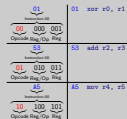
Opcode is 00: `xor`; 01: `add`; 10: `mov`. Register #s in lower fields.

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

- Synthesizing Machine-Code Programs

- Synthesizing Machine-Code Programs

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

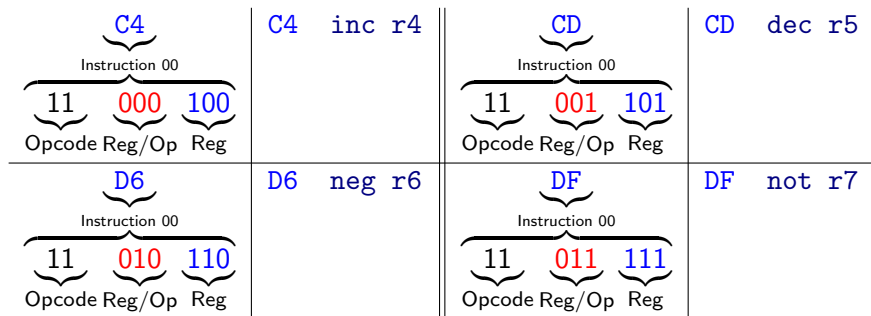
Opcode is 00: xor; 01: add; 10: mov. Register #s in lower fields.

The next few slides just overview the machine code encoding. This is just the easiest way I could think of to encode the instructions. You don't need to pay much attention, and we'll go quickly. Basically we divide the opcode bytes into three fields. For the instructions with two registers, we just use the top field to specify the operation, and the other fields to specify the register numbers.

Synthesizing Machine-Code Programs

SIMPLE Machine Language

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



Opcode field is 11.

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

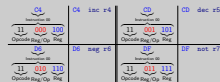
Register number in lowest field.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Machine-Code Programs

Synthesizing Machine-Code Programs

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

Opcode field is 11.

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

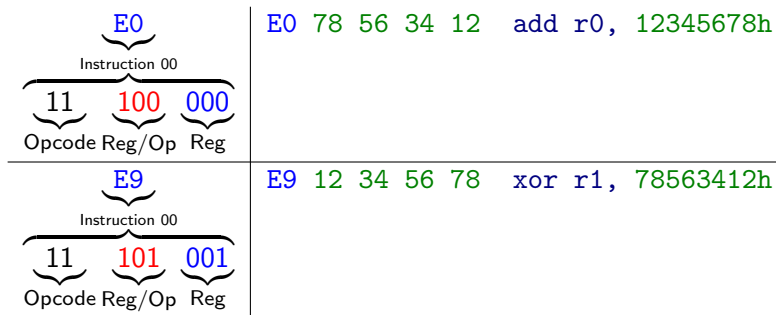
Register number in lowest field.

For the unary instructions, we only need one field to specify the register number. So we fix the top fields to 11, use the low field for the register number, and the middle field for the operation.

Synthesizing Machine-Code Programs

SIMPLE Machine Language

Machine-code encoding for binary `reg/imm32` SIMPLE instructions.



Opcode field is 11.

Middle 3 are 100: `add`; 101: `xor`; 110: `and`; 111: `or`.

Register number in lowest field. Constant follows opcode byte.

Synesthesia: A Modern Approach to Shellcode Generation

- Symbolic Program Synthesis

- Synthesizing Machine-Code Programs

- Synthesizing Machine-Code Programs

Machine-code encoding for binary `reg/imm32` SIMPLE instructions.

Opcode field is 11.

Middle 3 are 100: add; 101: xor; 110: and; 111: or.

Register number in lowest field. Constant follows opcode byte.

For the binary instructions with constants, it's the same as the unary case, except a 32-bit constant follows the opcode byte. Easy.

Synthesizing Machine-Code Programs

SIMPLE Machine Language, Decoding Opcodes

```
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 SIMPLEMENTC byte.

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Machine-Code Programs

Synthesizing Machine-Code Programs

Synthesizing Machine-Code Programs
SIMPLC Machine Language: Decoding Opcode

```
Simple DecodeOpcode(int firstByte)
int topTwo = (firstByte >> 4) & 3;
int midThree = (firstByte >> 3) & 7;

if(topTwo != 0x1)
    return EnumReg + topTwo;

return EnumReg + midThree;
```

This function decodes the opcode from a SIMPLC byte.

And that's it. This function looks at an opcode byte and returns the associated enumeration element. It's easy, and the details aren't very important.

Synthesizing Machine-Code Programs

SIMPLE Machine Language, Decoding Instructions

```
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 SIMPLMEC bytes.

1. Fetch the opcode ◀.
2. Determine the instruction's length ◀.
3. Extract the register numbers ◀.
4. Extract the 32-bit constant ◀.

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Machine-Code Programs

Synthesizing Machine-Code Programs

Synthesizing Machine-Code Programs
SIMPLC Machine Language, Decoding Instructions

```
void Decode(char *bytes, int *ip, int *length, Instruction *ins)  
{  
    ▶ ins->op = DecodeOpcode(bytes[*ip]);  
    ▶ *length = ins->op < AddrRegImm ? 1 : 5;  
    ▶ ins->lhsRegNum = (firstByte>>3) & 7;  
    ▶ ins->rhsRegNum = firstByte & 7;  
    ▶ if(ins->op > AddrRegImm)  
        ▶ ins->lhsRegNum = ins->rhsRegNum;  
    ▶ ins->imm32 = *(uint32 *)(&bytes[*ip+1]);  
}
```

This function decodes an instruction from SIMPLC bytes.

1. Fetch the opcode ◀
2. Determine the instruction's length ▶
3. Extract the register numbers ◀
4. Extract the 32-bit constant ▶

The disassembler function is a bit uglier. It gets the opcode, calculates the length of the instruction, figures out the register numbers, and decodes the 32-bit constant. The details aren't important.

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;
```

```
}
```

Synthesizing SIMPLMC Programs

```
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;
```

```
}
```

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

Synesthesia: A Modern Approach to Shellcode Generation

Symbolic Program Synthesis

Synthesizing Machine-Code Programs

Synthesizing Machine-Code Programs

```
Synthesizing Machine-Code Programs
Comparison with Synthesizing ARM Programs

Synthesizing SIMPLE Programs
Instruction i1, i2; ◀
state *f(state *in)
{
    state *s1=EmulateOne(in, i1);
    state *s2=EmulateOne(s1, i2);
    return s2;
}

Synthesizing SIMPLMC Programs
char nc[256]; ◀
state *f(state *in)
{
    int i1, i2; ◀
    Instruction i1, i2; ◀
    Decode(nc, 0, &i1, &i2); ◀
    Decode(nc, i1, &i2, &i3); ◀
    state *s1=EmulateOne(in, i1);
    state *s2=EmulateOne(nc, i2,
    return s2;
}

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

And now we can synthesize machine code programs. This is the same slide from before; now it should make sense. We just use an array of machine code bytes as our components, decode two instructions, let them act on the state, and return the final state. Now we can synthesize machine code programs. Rather than give an example immediately, let's just jump to encoding restrictions.

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.

Symbol	Description	Contents
$\exists \text{COMPONENTS}$	Exists components	$\exists mc \in \mathbf{Array}[\mathbf{BV}[8]] \rightarrow \mathbf{BV}[8]$
$\forall \text{INPUTS}$	For all inputs	$\forall in \in \mathbf{State}$
ϕ_{Program}	Program constraint	...
$\phi_{\text{Functionality}}$	Functionality constraint	...

Encoding Restrictions

Overview

Recall from before that our formulas have roughly this structure.

Symbol	Description	Contents
$\exists \overline{\text{COMPONENTS}}$	Exists components	$\exists \text{mc} \in \mathbf{Array}[\mathbf{BV}[8]] \rightarrow \mathbf{BV}[8]$
$\forall \overline{\text{INPUTS}}$	For all inputs	$\forall \text{in} \in \mathbf{State}$
ϕ_{Program}	Program constraint	...
$\phi_{\text{Functionality}}$	Functionality constraint	...
ϕ_{Encoding}	Encoding constraint	<i>Shown next</i>

We begin by adding **encoding restrictions**; say, no NULL bytes.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

Recall from before that our formulas have roughly this structure.

Symbol	Description	Contents
COMPONENTS	Exists components	$\exists aa \in \text{Array}[BV[B]] \rightarrow BV[B]$
INPUTS	For all inputs	$\forall ia \in \text{State}$
Program	Program constraint	...
Functionality	Functionality constraint	...
Encoding	Encoding constraint	Shown next

We begin by adding **encoding restrictions**, say, no NULL bytes.

From before, all of our synthesis formulas have this same structure.

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

Recall from before that our formulas have roughly this structure.

Symbol	Description	Contents
\exists COMPONENTS	Exists components	$\exists \text{aa} \in \text{Array}[\text{BV}[\text{B}]] \rightarrow \text{BV}[\text{B}]$
\forall INPUTS	For all inputs	$\forall \text{ia} \in \text{State}$
Program	Program constraint	...
Functionality	Functionality constraint	...
Encoding	Encoding constraint	Shown next

We begin by adding **encoding restrictions**, say, no NULL bytes.

Now let's add encoding constraints to the picture.

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:

English

Mathematics

For all array indices i

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:

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[8]$.

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:

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[8]$.
Within our synthesized machine code	

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:

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[8]$.
Within our synthesized machine code	$i < \mathbf{len1} + \mathbf{len2} \Rightarrow$

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:

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[8]$.
Within our synthesized machine code	$i < \text{len1} + \text{len2} \Rightarrow$
Machine code byte $\#i$ is not $0x00$	

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:

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[8]$.
Within our synthesized machine code	$i < \text{len1} + \text{len2} \Rightarrow$
Machine code byte $\#i$ is not $0x00$	$\text{mc}[i] \neq 0x00$

$$\phi_{\text{Non-NULL}} := [\forall i : \mathbf{BV}[8] \cdot i < \text{len1} + \text{len2} \Rightarrow \text{mc}[i] \neq 0x00]$$

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

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.

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[\mathbf{B}]$
Within our synthesized machine code	$i < 1\text{mb}1 + 1\text{mb}2 \rightarrow$
Machine code byte $a[i]$ is not 0x00	$a[i] \neq 0x00$
$\phi_{\text{No NULL}} := [\forall i : \mathbf{BV}[\mathbf{B}] \cdot i < 1\text{mb}1 + 1\text{mb}2 \Rightarrow a[i] \neq 0x00]$	

Now we show how to specify an encoding restriction. Let's say we don't want any NULL bytes in our solution. Here's how to write that mathematically. We say that, for any array index i

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

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.

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[\mathbf{B}]$
Within our synthesized machine code	$i < 1\text{mb}1 + 1\text{mb}2 \rightarrow$
Machine code byte $a[i]$ is not 0x00	$a[i] \neq 0x00$

$$\psi_{\text{No NULL}} := [\forall i : \mathbf{BV}[\mathbf{B}]. i < 1\text{mb}1 + 1\text{mb}2 \Rightarrow a[i] \neq 0x00]$$

As before, we use a universal quantifier “for all”

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

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.

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[\mathbf{B}]$
Within our synthesized machine code	$i < 1\text{mb}1 + 1\text{mb}2 \rightarrow$
Machine code byte $\text{# } i$ is not 0x00	$\text{ac}[i] \neq 0x00$

$$\psi_{\text{No NULL}} := [\forall i : \mathbf{BV}[\mathbf{B}]. i < 1\text{mb}1 + 1\text{mb}2 \Rightarrow \text{ac}[i] \neq 0x00]$$

If the array index is within our machine code

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

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.

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[\mathbf{B}]$
Within our synthesized machine code	$i < 1\text{mb} + 1\text{mb} \rightarrow$
Machine code byte $\text{# } i$ is not 0x00	$\text{ac}[i] \neq 0x00$

$$\psi_{\text{No NULL}} := [\forall i : \mathbf{BV}[\mathbf{B}]. i < 1\text{mb} + 1\text{mb} \Rightarrow \text{ac}[i] \neq 0x00]$$

That is, between the beginning of the array and the end of the last instruction

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

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.

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[\mathbf{B}]$
Within our synthesized machine code	$i < \text{len1} + \text{len2} \rightarrow$
Machine code byte $a[i]$ is not 0x00	$a[i] \neq 0x00$

$$\psi_{\text{No NULL}} := [\forall i : \mathbf{BV}[\mathbf{B}] \cdot i < \text{len1} + \text{len2} \Rightarrow a[i] \neq 0x00]$$

Then, the byte at that position is not zero

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

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.

English	Mathematics
For all array indices i	$\forall i \in \mathbf{BV}[\mathbf{B}]$
Within our synthesized machine code	$i < 1\text{msb} + 1\text{msb} \rightarrow$
Machine code byte $\#i$ is not 0x00	$ac[i] \neq 0x00$

$$\psi_{\text{No NULL}} := [\forall i : \mathbf{BV}[\mathbf{B}]. i < 1\text{msb} + 1\text{msb} \Rightarrow ac[i] \neq 0x00]$$

And then the total formula is shown at the bottom of the slide.

Encoding Restrictions

More Examples

These examples are all $[\forall i : \mathbf{BV}[8] \cdot i < \text{len1} + \text{len2} \Rightarrow \phi_{\text{Byte}}]$,
where ϕ_{Byte} is \dots :

Encoding Restriction	ϕ_{Byte}
No NULL Bytes	$mc[i] \neq 0x00$
No '%' Bytes	$mc[i] \neq 0x25$
All ASCII are Uppercase	$\neg(mc[i] \geq 0x61 \wedge mc[i] \leq 0x7A)$
All Bytes Printable	$(mc[i] \geq 0x21 \wedge mc[i] \leq 0x7F)$
All Bytes Alphanumeric	$(mc[i] \geq 0x30 \wedge mc[i] \leq 0x39) \vee$ $(mc[i] \geq 0x41 \wedge mc[i] \leq 0x5A) \vee$ $(mc[i] \geq 0x61 \wedge mc[i] \leq 0x7A)$

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Encoding Restrictions

Encoding Restrictions

These examples are all $[r_i : BV[B]] : i < len + len0 \Rightarrow \phi_{BV}$,
where ϕ_{BV} is ...:

Encoding Restriction	ϕ_{BV}
No NULL Bytes	$ac[i] \neq 0x00$
No '\x' Bytes	$ac[i] \neq 0x5c$
All ASCII are Uppercase	$\neg(ac[i] \geq 0x61 \wedge ac[i] \leq 0x7a)$
All Bytes Printable	$(ac[i] \geq 0x20 \wedge ac[i] \leq 0x7f)$
All Bytes Alphanumeric	$(ac[i] \geq 0x61 \wedge ac[i] \leq 0x7a) \vee (ac[i] \geq 0x41 \wedge ac[i] \leq 0x5a)$

Now, it's easy to formalize the other examples we gave at the beginning of the presentation. Those ones all have roughly the same structure as the "non-NULL" example we just saw. Except they exclude other values, or specify ranges of legal values. So we have non-NULL, no percentage character, no lowercase ASCII letters, all bytes are printable, all bytes are alphanumeric. Simple!

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.

$$[\forall i : \mathbf{BV}[8] \cdot i < \text{len1} + \text{len2} - 1 \Rightarrow (\text{mc}[i] <= \text{mc}[i+1])]]$$

Encoding Restrictions

More Complex Examples: Bytes Must Increase

Let's say our shellcode bytes must monotonically increase.

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

Let's say our shellcode bytes must strictly increase.

Encoding Restrictions

More Complex Examples: Bytes Must Increase

Let's say our shellcode bytes must monotonically increase.

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

Let's say our shellcode bytes must strictly increase.

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

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

Let's say our shellcode bytes must monotonically increase.

$$[? : \mathbf{BV}[8] - 1 < 1\text{ms} + 1\text{ms}2 - 1 \Rightarrow \text{nc}[1] \leftarrow \text{nc}[1+1]]$$

Let's say our shellcode bytes must strictly increase.

$$[? : \mathbf{BV}[8] - 1 < 1\text{ms} + 1\text{ms}2 - 1 \Rightarrow \text{nc}[1] < \text{nc}[1+1]]$$

Now let's show some more exotic examples. Let's say the bytes must monotonically increase. That means each byte is less than or equal to the following byte.

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

Let's say our shellcode bytes must monotonically increase.

$$[\forall i : \text{BV}[i] - 1 < 1\text{ms} + 1\text{ms}2 - 1 \Rightarrow \text{ac}[i] \leftarrow \text{ac}[i+1]]$$

Let's say our shellcode bytes must strictly increase.

$$[\forall i : \text{BV}[i] - 1 < 1\text{ms} + 1\text{ms}2 - 1 \Rightarrow \text{ac}[i] < \text{ac}[i+1]]$$

This is easy to formalize; just use a less-than-or-equal-to operator to compare adjacent bytes.

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

Let's say our shellcode bytes must monotonically increase.

$$[? : \mathbf{BV}[8] - 1 < 1\text{ms} + 1\text{ms}2 - 1 \Rightarrow \text{ac}[1] \leftarrow \text{ac}[1+1]]$$

Let's say our shellcode bytes must strictly increase.

$$[? : \mathbf{BV}[8] - 1 < 1\text{ms} + 1\text{ms}2 - 1 \Rightarrow \text{ac}[1] < \text{ac}[1+1]]$$

Or if we wanted strictly increasing bytes; no adjacent duplicates.

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

Let's say our shellcode bytes must monotonically increase.

$$[\forall i : \text{BV}[i] - i < 1\text{msl} + 1\text{ms2} - 1 \Rightarrow \text{ac}[i] \leftarrow \text{ac}[i+1]]$$

Let's say our shellcode bytes must strictly increase.

$$[\forall i : \text{BV}[i] - i < 1\text{msl} + 1\text{ms2} - 1 \Rightarrow \text{ac}[i] < \text{ac}[i+1]]$$

Easy; just use less-than.

Encoding Restrictions

More Complex Examples: No Duplicate Bytes

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

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

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

Let's say all the bytes in our shellcode must differ. Use two “forall” quantifiers over array indices, and say that the values at each of the array indices must differ. Basically, every byte must differ from every byte before it.

Encoding Restrictions

More Complex Examples: No Duplicate Bytes

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

$$[\forall i : \mathbf{BV}[8], j : \mathbf{BV}[8] \cdot i < \text{len1} + \text{len2} - 1 \wedge j < i \Rightarrow (\text{mc}[j] \neq \text{mc}[i])]]$$

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

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

$$[i: BV[B], j: BV[B] \cdot i < len1 + len2 - 1 \wedge j < i \Rightarrow len\{i\} \neq len\{j\}]$$

Let's say all the bytes in our shellcode must differ. Use two “forall” quantifiers over array indices, and say that the values at each of the array indices must differ. Basically, every byte must differ from every byte before it.

Encoding Restrictions

More Complex Examples: Alternating Even/Odd

Let's say our shellcode must alternate between even and odd bytes.

Encoding Restrictions

More Complex Examples: Alternating Even/Odd

Let's say our shellcode must alternate between even and odd bytes.

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

Encoding Restrictions

More Complex Examples: Alternating Even/Odd

Let's say our shellcode must alternate between even and odd bytes.

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

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

Encoding Restrictions

More Complex Examples: Alternating Even/Odd

Let's say our shellcode must alternate between even and odd bytes.

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

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

$$\text{mc}[0] \&1 == 0$$

Synesthesia: A Modern Approach to Shellcode Generation

└─ Extensions

└─ Encoding Restrictions

└─ Encoding Restrictions

Let's say our shellcode must alternate between even and odd bytes.

```
[0: BV[32] - 1 < 1aaS + 1aaS - 1 => (a[1]*a[1+1])a1 == 1]
```

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

```
a[0]a1 == 0
```

If we wanted our bytes to alternate between even and odd:

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

Encoding Restrictions

More Complex Example: Alternating Even/Odd

Let's say our shellcode must alternate between even and odd bytes.

```
[0: BV[32] - 1 < 1aaS + 1aaS - 1 => (a[1]*a[1+1])a1 == 1]
```

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

```
a[0]a1 == 0
```

We just say that the lowest bits of adjacent bytes differ.

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

Encoding Restrictions

More Complex Example: Alternating Even/Odd

Let's say our shellcode must alternate between even and odd bytes.

```
[0: BV[32] - 1 < 1aaS + 1aaS - 1 => (a[1]*a[1+1])a1 == 1]
```

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

```
a[0]a1 == 0
```

And, if we needed our first byte to be even

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Encoding Restrictions

└ Encoding Restrictions

Encoding Restrictions

More Complex Example: Alternating Even/Odd

Let's say our shellcode must alternate between even and odd bytes.

```
[0: BV[32] - 1 < 1aaS + 1aaS - 1 => (a[1]*a[1+1])a1 == 1]
```

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

```
a[0]a1 == 0
```

We just specify that as a constraint on the lowest bit of the first byte.

Encoding Restrictions

More Complex Examples: All words are Prime Numbers

A word w is prime if only 1 and w divides evenly into it. I.e.:

$$\forall d : \mathbf{BV}[16] \cdot (2 \leq d \ \&\& \ d < w) \Rightarrow w \% d \neq 0$$

Synesthesia: A Modern Approach to Shellcode Generation

└─ Extensions

└─ Encoding Restrictions

└─ Encoding Restrictions

A word w is prime if only 1 and w divides evenly into it. I.e.:

$\forall d: \text{BV}[d] \rightarrow (d \rightarrow d \text{ or } d \leftarrow w) \rightarrow w \Sigma d \neq 0$

We can do crazier things than that; we can specify that the **words** of our shellcode must be prime numbers. We can express primality by saying only one and the number itself divide into a given word leaving no remainder.

Encoding Restrictions

Solutions

Interesting solutions to `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 `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`

Extensions

Encoding Restrictions

Encoding Restrictions

```

Interesting solutions to eax == 0x0
-----
'First byte of each instruction = 0x0, all bytes non-NUL'
00 CA 01 48 FE  or eax, 0xFAB01CAh
10 CA 16 B3 5A  sbb eax, 5A5316CAh
19 CA          sbb eax, eax

```

```

Interesting solutions to eax == 0x12345678
-----
Alternating even/odd
80 7D 5D 5D 7A  mov eax, 7A5D5D7Ah
26 FC F7 34 17  and eax, 1734F7FCb
-----
Alternating even/odd (first byte odd)
81 C8 7D 5D 5D 7A  or  eax, 5D5D5D7Ah
81 E9 7D 5D 7D 92  and  eax, 927D5D7Dh
26 78 57 36 13  and  eax, 13365778h

```

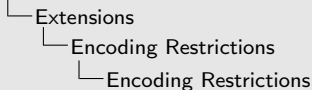
Here are a couple of interesting examples I came up with while playing with input restrictions. The one at the top sets `eax` to `0`, with some onerous character restrictions. Since it can't use the `sub` instruction, it uses the `sbb` instruction instead. But performing an `sbb` of a register with itself will either result in a value of `0` or `-1` depending on the carry flag. So first, it ORs `eax` with some large value, and then subtracts some smaller value. By doing this, it's able to guarantee that the carry flag will always be clear by the time the `sbb eax, eax` instruction executes. I might not have come up with that myself. Note that this is fully automated; the system has not been programmed to know this trick in advance.

The second examples set `eax` to `12345678h` using alternating even and odd bytes. I chose that constant since each byte is even. It sets `eax` to a value that alternates between even and odd bytes, and then ANDs with a constant whose bytes alternate in the opposite order. The second example does something similar, but needs two `and` instructions.

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.



- 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.

So you should get the picture that we can model a lot of exotic encoding restrictions using this technique. Anything that can be modelled using a first-order formula can be represented. And, in fact, if we collect an execution trace of the program, it implicitly contains the restrictions and transformations without us having to model them – but that method does have some limitations with completeness.

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.

Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└└ Synthesis of Equivalent Snippets

└└└ Synthesis of Equivalent Snippets

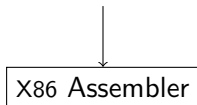
- Suppose we already have machine code that does what we want, but it doesn't satisfy the encoding restrictions.
- Simply express the input/output behavior of that code, and use that as the functional constraint.

We can also use this technique to find equivalent sequences to machine-code snippets that we already have. We do this using functional constraints: we express the input/output behavior of our existing sequence, and use that to specify the behavior of the thing we want to synthesize.

Synthesis of Equivalent Snippets

Step #1: Assemble the Code

```
mov eax, 1  
mov ebx, 2  
mov ecx, 3
```



```
B8 01 00 00 00  
BB 02 00 00 00  
B9 03 00 00 00
```

First, assemble the code you wish to replace.

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesis of Equivalent Snippets

Synthesis of Equivalent Snippets

Synthesis of Equivalent Snippets
Step 2.1: Assemble the Code

```
mov eax, 1  
mov ebx, 2  
mov ecx, 3
```

X86 Assembler

```
B8 01 00 00 00  
BB 02 00 00 00  
BC 03 00 00 00
```

First, assemble the code you wish to replace.

We begin by simply assembling the X86 instructions into machine code bytes.

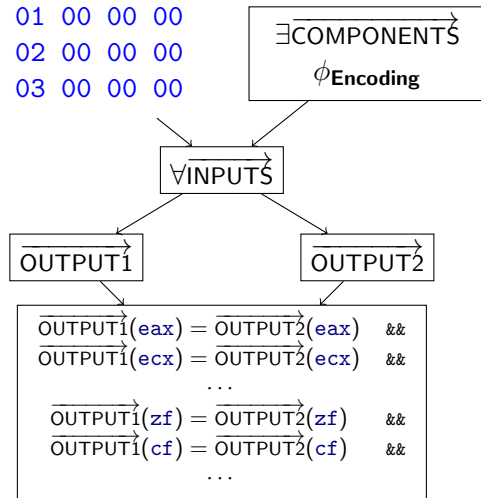
Synthesis of Equivalent Snippets

Step #2: Synthesize Replacement

B8 01 00 00 00

BB 02 00 00 00

B9 03 00 00 00



Next, synthesize equivalent code within the encoding.

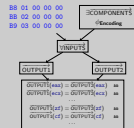
If desired, omit irrelevant registers or flags from the functionality constraint.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesis of Equivalent Snippets

Synthesis of Equivalent Snippets



This slide shows visually what I just said: we take existing machine code, express its input/output behavior, and use that as the functional constraint in synthesizing machine code under the encoding restriction. If we want, we can loosen the restriction a bit by saying that not all flags must match, or that it is allowed to overwrite the values of certain registers, etc.

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



Finding the Shortest Program

Conceptually

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

English		Mathematics
Is there a best program	$\exists \overrightarrow{BEST}$	
Which is good, and	$Good(\overrightarrow{BEST})$	

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \overrightarrow{BEST}$
Which is good, and	$Good(\overrightarrow{BEST})$
For all other programs	$\forall \overrightarrow{OTHER}$

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \overrightarrow{BEST}$
Which is good, and	$Good(\overrightarrow{BEST})$
For all other programs	$\forall \overrightarrow{OTHER}$
If the other program is good	$Good(\overrightarrow{OTHER})$

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \overrightarrow{BEST}$
Which is good, and	$Good(\overrightarrow{BEST})$
For all other programs	$\forall \overrightarrow{OTHER}$
If the other program is good	$Good(\overrightarrow{OTHER})$
The other is at least as long	$\Rightarrow Length(\overrightarrow{BEST}) \leq Length(\overrightarrow{OTHER})$

- ▶ Here we quantify **over all solutions**.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \text{BEST}$
Which is good, and	$\text{Good}(\overline{\text{BEST}})$
For all other programs	$\forall \text{OTHER}$
If the other program is good	$\text{Good}(\overline{\text{OTHER}})$
The other is at least as long	$\Rightarrow \text{Length}(\overline{\text{BEST}}) \leq \text{Length}(\overline{\text{OTHER}})$

- Here we quantify **over all solutions**.

OK, let's say we want the shortest solution to a given synthesis problem. To be brief, let's just say that a program is **good** if it satisfies the synthesis constraints. Now, we are looking for the shortest good program.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \text{BEST}$
Which is good, and	$\text{Good}(\overline{\text{BEST}})$
For all other programs	$\forall \text{OTHER}$
If the other program is good	$\text{Good}(\overline{\text{OTHER}})$
The other is at least as long	$\Rightarrow \text{Length}(\overline{\text{BEST}}) \leq \text{Length}(\overline{\text{OTHER}})$

- Here we quantify **over all solutions**.

So first, we have “exists best”. This is our shortest program.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \overline{\text{BEST}}$
Which is good, and	$\text{Good}(\overline{\text{BEST}})$
For all other programs	$\forall \overline{\text{OTHER}}$
If the other program is good	$\text{Good}(\overline{\text{OTHER}})$
The other is at least as long	$\Rightarrow \text{Length}(\overline{\text{BEST}}) \leq \text{Length}(\overline{\text{OTHER}})$

- Here we quantify **over all solutions**.

Of course, “best” must satisfy the synthesis constraints, so we say, “best is good”.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \text{BEST}$
Which is good, and	$\text{Good}(\overline{\text{BEST}})$
For all other programs	$\forall \text{OTHER}$
If the other program is good	$\text{Good}(\overline{\text{OTHER}})$
The other is at least as long	$\Rightarrow \text{Length}(\overline{\text{BEST}}) \leq \text{Length}(\overline{\text{OTHER}})$

- Here we quantify **over all solutions**.

Now, for every other program – we use a “for all” quantifier to quantify over all other programs.

2016-11-08

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \text{BEST}$
Which is good, and	$\text{Good}(\overline{\text{BEST}})$
For all other programs	$\forall \text{OTHER}$
If the other program is good	$\text{Good}(\overline{\text{OTHER}})$
The other is at least as long	$\Rightarrow \text{Length}(\overline{\text{BEST}}) \leq \text{Length}(\overline{\text{OTHER}})$

- Here we quantify **over all solutions**.

If the other program is good

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Finding the Shortest Program

Conceptually

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

English	Mathematics
Is there a best program	$\exists \text{BEST}$
Which is good, and	$\text{Good}(\overline{\text{BEST}})$
For all other programs	$\forall \text{OTHER}$
If the other program is good	$\text{Good}(\overline{\text{OTHER}})$
The other is at least as long	$\Rightarrow \text{Length}(\overline{\text{BEST}}) \leq \text{Length}(\overline{\text{OTHER}})$

- Here we quantify **over all solutions**.

Then the length of the best program is less than or equal to the other program. So, succinctly, out of all the good programs, this formula finds the shortest one.

Finding the Shortest Program

Conceptually

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

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

Finding the Shortest Program

Conceptually

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

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

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

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

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Here is a shortest solution to $(\text{functionality}=\text{increment-r0})$:

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

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

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

So if we applied that idea to the increment-`r0` functionality constraint, the shortest two-instruction program would be two bytes. That makes sense; each instruction is at least one byte, and we specified that our solution consists of two instructions.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Finding the Shortest Program

Finding the Shortest Program

Finding the Shortest Program
Conceptually

Here is a shortest solution to $\text{!functionality-increase-r5}$:

```
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 or r0, 0
```

We could also tweak the constraint a bit to target the longest program: this one says, for all good programs, the best solution is at least as long. So now we get a ten-byte solution; two instructions, five bytes apiece.

Extensions

Encoding Restrictions

Synthesis of Equivalent Snippets

Finding the Shortest Program

Synthesizing Decoders

Input State Preconditions

Integration with Exploit Generation

Synthesizing Decoders

Overview on Loops

We shall now automate synthesis of decoder loops.

```
    ; 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.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

We shall now automate synthesis of decoder loops.

```
0: Initialize counter
@loop:
0: Get encoded byte
0: Decode encoded byte
0: Store decoded byte
0: Decrement counter
0: 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.

Now let's get a little more ambitious. Let's try to automatically encode, and automatically generate a decoder, for a given blob of instructions. In order to do that, we're going to need a loop. So first, we'll review how to deal with loops when using logic-based methods like SMT solvers.

We're going to need two ingredients: a **loop invariant**, which specifies how the loop behaves; and a **loop variant**, which proves that the loop terminates.

Synthesizing Decoders

Crash Course on Treating Loops Formally

- ▶ Let's review loop invariants and variants with an example.
- ▶ We show that `max` terminates with the greatest element of `arr`.

```
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.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

Synthesizing Decoders

Crash Course on Treating Loops Formally

- ▶ Let's review loop invariants and variants with an example.
- ▶ We show that `max` terminates with the greatest element of `arr`.

```
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.

We'll talk about loop invariants and loop variants in terms of this simple example. This function `max` finds the greatest value in a given array. It's probably pretty easy to convince yourself of that just by looking at it, but we'll be formal about it.

The function starts by taking the first element of the array into the variable `m`, then for every iteration, if the current value is bigger than the biggest one we've seen so far, it updates `m` to contain that value, and returns the final value of `m`.

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.

```
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:



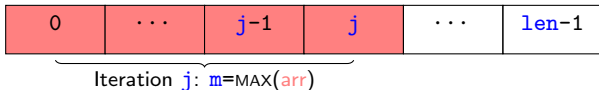
Synthesizing Decoders

Loop Invariants

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

```
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:
 - ▶ After iteration # j , $m = \text{MAX}(\text{arr}, 0, j)$.



Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

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.

```
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:
- After iteration #j, $m = \text{MAX}\{\text{arr}[0..j]\}$.



A loop invariant contains two parts: it says that, if some property is true before a loop iteration, then the property is still true after the iteration.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

- A **loop invariant** says that:

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

```
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:
- After iteration #j, $m = \text{MAX}(\text{arr}[0..j])$.



So, specifically, we want to say that: if the value of m contains the greatest value of the sub-array before an iteration (i.e., if m contains the greatest value from the blue shaded region before the iteration)

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

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.

```
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:
- After iteration #j, $m = \text{MAX}(\text{arr}[0..j])$.



Then, after the iteration, m contains the greatest value from the region that is one bigger, i.e., the greatest value from the **red** shaded region.

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 , $m = \text{MAX}(\text{arr}, 0, j)$.

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

- ▶ The loop invariant is true because:

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 , $m = \text{MAX}(\text{arr}, 0, j)$.

```
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 $m = \text{MAX}(\text{arr}, 0, j-1) = \text{MAX}(\text{arr}, 0, j)$.

Synthesizing Decoders

Loop Invariants

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

```
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 $\text{arr}[j]$.

Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

Synthesizing Decoders

Loop Invariants

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

for (int i=1; i<n; i++)
 if (arr[i] > a)
 a = arr[i];

```
- The loop invariant is true because:
    - If  $\text{arr}[j] \leq a$ , then:
      - $\text{arr}[j]$  is not greater than some previous element ( $a$ ).
      - Thus the existing  $a = \text{MAX}(\text{arr}, 0, j-1) = \text{MAX}(\text{arr}, 0, j)$ .
    - If  $\text{arr}[j] > a$ , then:
      - $\text{arr}[j]$  is greater than all previous elements.
      - Thus  $\text{arr}[j] = \text{MAX}(\text{arr}, 0, j)$ , and  $a$  becomes  $\text{arr}[j]$ .

So this slide begins by repeating the loop invariant from the last slide. We can show that the loop invariant is true, because: if, before the iteration,  $m$  contains the largest value from the first  $i$  values of the array, then, one of two things is going to happen in the loop body.

## Synesthesia: A Modern Approach to Shellcode Generation

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

## Synthesizing Decoders

## Loop Invariants

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

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

- The loop invariant is true because:
  - If  $arr[j] \leq a$ , then:
    - $arr[j]$  is not greater than some previous element (a).
    - Thus the existing  $a = \text{MAX}(arr.D, j-1) = \text{MAX}(arr.D, j)$ .
  - If  $arr[j] > a$ , then:
    - $arr[j]$  is greater than all previous elements.
    - Thus  $arr[j]$  is  $\text{MAX}(arr.D, j)$ , and  $a$  becomes  $arr[j]$ .

Either the value of the array at the current position is not bigger than the previous maximum value, in which case the previous maximum value is still the maximum



## Synesthesia: A Modern Approach to Shellcode Generation

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

## Synthesizing Decoders

## Loop Invariants

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

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

- The loop invariant is true because:
  - If  $arr[j] \leq a$ , then:
    - $arr[j]$  is not greater than some previous element ( $a$ ).
    - Thus the existing  $a = \text{MAX}(arr.D, j-1) = \text{MAX}(arr.D, j)$ .
  - If  $arr[j] > a$ , then:
    - $arr[j]$  is greater than all previous elements.
    - Thus  $arr[j]$  is  $\text{MAX}(arr.D, j)$ , and  $a$  becomes  $arr[j]$ .

Or, the current value is bigger than the previous maximum, meaning it is the maximum value up to that point, and in which case we update  $m$  to contain that value.

# Synthesizing Decoders

## Loop Invariants

```
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,  $m = \text{MAX}(\text{arr}, 0, 0)$ .
- ▶ After every iteration #j,  $m = \text{MAX}(\text{arr}, 0, j)$ .
  - ▶ Specifically, after iteration #len-1,  $m = \text{MAX}(\text{arr})$ .
- ▶ Thus the code above correctly computes the array maximum.
  - ▶ “Partially correct” because we have not yet proved termination.

## Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└ Synthesizing Decoders

└ Synthesizing Decoders

Synthesizing Decoders  
Loop Invariants

```
1 int a = arr[0];
2 for(int i=1;i<len;++i)
3 if(arr[i] > a)
4 a = arr[i];
5 return a;
```

- Because of line #1, before iteration #1,  $a = \text{MAX}(\text{arr}, 0, 0)$ .
- After every iteration # $j$ ,  $a = \text{MAX}(\text{arr}, 0, j)$ .
  - Specifically, after iteration # $\text{len}-1$ ,  $a = \text{MAX}(\text{arr})$ .
- Thus the code above correctly computes the array maximum.
  - "Partially correct" because we have not yet proved termination.

So, before the loop executes,  $m$  contains the maximum value from the sub-array of length 1. Then, after each iteration  $j$ ,  $m$  contains the maximum value from the sub-array of length  $j+1$ . Therefore, after iteration  $\text{len}-1$ ,  $m$  contains the maximum value of the entire array. So the code correctly computes the maximum value. We call that “partially correct”: if it terminates, it does so with the correct value. But we also need to show that the loop terminates when it reaches the end of the array.

# Synthesizing Decoders

## Loop Variants

- ▶ 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.

└ Extensions

└─┬ Synthesizing Decoders

└─┬ Synthesizing Decoders

- 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  $n - 1$ .
  - The "amount of work remaining" cannot decrease forever.
- The **variant function** gives the amount of work remaining.

Now we move on to loop variants. Loop variants describe the amount of work left to compute, and show that the amount of work always decreases, and can't decrease forever.

# Synthesizing Decoders

## Loop Variants

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

| # | Code                                   | $i$ | $v(i)$ |
|---|----------------------------------------|-----|--------|
| 1 | <code>for(int i=1;i&lt;len;++i)</code> | 1   | 9      |
| 2 | <code>  if(arr[i] &gt; m)</code>       |     |        |
| 3 | <code>    m = arr[i];</code>           |     |        |

Assuming  $\text{len} = 10$

# Synthesizing Decoders

## Loop Variants

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

| # | Code                                   | $i$ | $v(i)$ |
|---|----------------------------------------|-----|--------|
| 1 | <code>for(int i=1;i&lt;len;++i)</code> | 2   | 8      |
| 2 | <code>  if(arr[i] &gt; m)</code>       |     |        |
| 3 | <code>    m = arr[i];</code>           |     |        |

Assuming  $\text{len} = 10$

# Synthesizing Decoders

## Loop Variants

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

| # | Code                                   | $i$ | $v(i)$ |
|---|----------------------------------------|-----|--------|
| 1 | <code>for(int i=1;i&lt;len;++i)</code> | 9   | 1      |
| 2 | <code>  if(arr[i] &gt; m)</code>       |     |        |
| 3 | <code>    m = arr[i];</code>           |     |        |

Assuming  $\text{len} = 10$



# Synthesizing Decoders

## Loop Variants

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

| # | Code                                   | $i$ | $v(i)$ |
|---|----------------------------------------|-----|--------|
| 1 | <code>for(int i=1;i&lt;len;++i)</code> | 10  | 0      |
| 2 | <code>  if(arr[i] &gt; m)</code>       |     |        |
| 3 | <code>    m = arr[i];</code>           |     |        |

Assuming `len = 10`

- ▶ When  $i = \text{len}$  ( $v(i) = 0$ ), the loop terminates.

## Synesthesia: A Modern Approach to Shellcode Generation

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

- Define a variant  $v(i)$  =  $len-i$  (number of iterations left).

| # | Code                                     | $i$ | $v(i)$ |
|---|------------------------------------------|-----|--------|
| 1 | <code>for(int i=1; i&lt;len; ++i)</code> | 10  | 0      |
| 2 | <code>if (arr[i] &gt; 0)</code>          |     |        |
| 3 | <code>w = arr[i];</code>                 |     |        |

Assuming `len = 10`

- When  $i=len$  ( $v(i)=0$ ), the loop terminates.

We specify loop variants in terms of a **variant function**. In this case, the variant function is `len-i`, the number of iterations remaining. Let's assume that `len` is 10. On the first iteration, there are 9 iterations remaining.

## Synesthesia: A Modern Approach to Shellcode Generation

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

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

| # | Code                     | i  | v(i) |
|---|--------------------------|----|------|
| 1 | for(int i=1; i<len; ++i) | 10 | 0    |
| 2 | if (arr[i] > n)          |    |      |
| 3 | n = arr[i];              |    |      |

Assuming  $\text{len} = 10$

- When  $i = \text{len}$  ( $v(i) = 0$ ), the loop terminates.

On the second iteration, there are 8 iterations remaining.

## Synesthesia: A Modern Approach to Shellcode Generation

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

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

| # | Code                     | i  | v(i) |
|---|--------------------------|----|------|
| 1 | for(int i=1; i<len; ++i) | 10 | 0    |
| 2 | if(arr[i] > 0)           |    |      |
| 3 | a = arr[i];              |    |      |

Assuming  $\text{len} = 10$

- When  $i = \text{len}$  ( $v(i) = 0$ ), the loop terminates.

On the ninth iteration, there is one iteration remaining.

## Synesthesia: A Modern Approach to Shellcode Generation

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

- Define a variant  $v(i) = len - i$  (number of iterations left).

| # | Code                     | i  | v(i) |
|---|--------------------------|----|------|
| 1 | for(int i=1; i<len; ++i) | 10 | 0    |
| 2 | if(arr[i] > 0)           |    |      |
| 3 | a = arr[i];              |    |      |

Assuming  $len = 10$

- When  $i=len$  ( $v(i)=0$ ), the loop terminates.

And, finally, on the 10th iteration, there are no iterations remaining. The loop test `i<len` is false; and so the loop terminates.

# Synthesizing Decoders

## Loop Variants

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

- ▶ Properties of variant  $v(i) = \text{len}-i$ :
  - ▶ 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  $\text{len}-1$  times (descent is finite).
  - ▶ When  $v(i)$  becomes 0, the loop terminates.
- ▶ Therefore, the loop executes a finite number of times ( $\text{len}-1$ ).
- ▶ Therefore, the loop terminates.
  - ▶ “Total correctness”: partial correctness plus termination.

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

```

1 for(iac = 1; i < len; ++i)
2 if(arr[i] > n)
3 a = arr[i];

```

- Properties of variant  $v(i) = len - i$ :
  - For every iteration,  $v(i) \geq 0$  (non-negative, minimum 0).
  - For each iteration  $i+1$ ,  $v(i+1) < v(i)$  ( $n$  decreases).
    - Because  $i$  increases on line  $\geq 1$ .
  - $v(i)$  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.

The fact that the loop has a variant function allows us to prove that it terminates. We can establish these properties pretty easily: the variant is always non-negative (greater than or equal to zero); it decreases on each iteration; it cannot decrease an infinite number of times; and the loop terminates when the variant reaches zero. Taken together, we have just proven that the loop executes  $len-1$  times, which means that it always terminates. So now we know our code is "totally correct", meaning, it does what we expect it to do, and it also terminates.

# Synthesizing Decoders

## Decoder Loop Invariants and Variants

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  $rC == 0$ )



# Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

## Synthesizing Decoders

Decoder Loop Invariants and Variants

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:**  $v(i) = 3m - i$ .

- Also, before iteration #0, some register  $rC$  is set to  $3m - v(0)$ .

**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  $slp$  is either:
    - The beginning of the loop (if  $rC \neq 0$ )
    - The instruction after the loop (if  $rC == 0$ )

So, if we want to synthesize code that has loops in it, we need to specify what the loop bodies are going to do, and also that the loops terminate. So we'll specify the synthesis behavior of our loop in terms of an invariant and a variant, and if we can synthesize code with those properties, then it's guaranteed to be a terminating loop that decrypts our shellcode. As a bonus, it'll encrypt the shellcode for free.

# Synthesizing Decoders

## Changes to SIMPLE's Machine State

### New SIMPLE Machine State

|           |                  |                                  |                            |
|-----------|------------------|----------------------------------|----------------------------|
| Registers | <code>eip</code> | Shellcode: Bytes <code>sc</code> | Pointer <code>scptr</code> |
|-----------|------------------|----------------------------------|----------------------------|

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`.

# Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

New SIMPLE Machine State

|           |                  |                                  |                            |
|-----------|------------------|----------------------------------|----------------------------|
| Registers | <code>*ip</code> | Shellcode: Bytes <code>sc</code> | Pointer <code>scptr</code> |
|-----------|------------------|----------------------------------|----------------------------|

The new machine state model contains:

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

We'll need to make some changes to SIMPLE to accommodate our demands. First, we'll need to model the shellcode as part of the machine state. We'll have an array called `sc` and a pointer into it called `scptr`. We'll also model the current `eip`.

# Synthesizing Decoders

## Changes to SIMPLE

---

### Instructions Added to SIMPLE:

---

```
getsbyte rX rX = sc[scptr]
putsbyte rX sc[scptr++] = rX
jnz rX, imm8 If rX \neq 0, jump to eip-imm8
```

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

---

### Instructions Removed from SIMPLE:

---

```
inc rX dec rX or rX, imm32
```

- ▶ Removed a few arithmetic instructions.
  - ▶ They didn't fit in the instruction set anymore.

## Extensions

## Synthesizing Decoders

## Synthesizing Decoders

---

Instructions Added to SIMPLE:

```
getachyte rX rX = &scptr
putachyte rX &scptr++ = rX
jnz rX, imm8 If rX ≠ 0, jump to wip+imm8
```

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

---

Instructions Removed from SIMPLE:

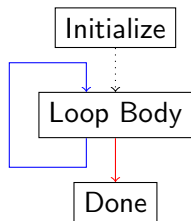
```
inc rX dec rX or rX, imm32
```

- Removed a few arithmetic instructions.
  - They didn't fit in the instruction set anymore.

Then we'll add some instructions to get and set shellcode bytes, and a jump-if-not-zero instruction for the end of our loop. The instruction that gets the current shellcode byte consults the shellcode pointer `scptr`; the instruction that sets the shellcode byte also uses the shellcode pointer, and also increments it after it executes. Since we're adding three instructions, we'll also need to get rid of three instructions; I chose a few of them more or less at random.

# Synthesizing Decoders

## Decoder Skeleton



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

# Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders



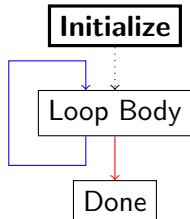
We synthesize two programs simultaneously:

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

Now our synthesis problem is really two related problems. We'll synthesize some code that executes before the loop, and the loop body. The code before the loop has to set some register to the length of the shellcode. The loop has to decrypt a byte of the shellcode, keep all of the other shellcode bytes the way they were, decrement that register containing the length, and then either branch back to the top of the loop or exit the loop if the counter has reached zero.

# Synthesizing Decoders

## Decoder Skeleton: Initialization



Synthesis formula for initialization block:

| English                              | Mathematics                       |
|--------------------------------------|-----------------------------------|
| Some register $rC$                   | $\exists ctr \in \mathbf{BV}[3]$  |
| Contains the length of the shellcode | $state_{After}[regs[ctr]] == len$ |

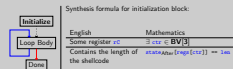


## Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders



The synthesis formula for the initialization is easy. We'll say, some register has to contain the length of the shellcode – and we don't care which register it is; let's call the register number `ctr` for counter.

# Synthesizing Decoders

## Decoder Skeleton: Loop Body

Synthesis formula for loop body:

If, before iteration:

---

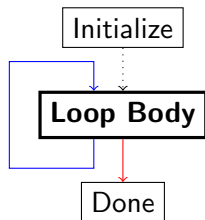
|                                              |                                         |
|----------------------------------------------|-----------------------------------------|
| <code>scptr</code> is within <code>sc</code> | <code>scptr &lt; len-1</code> &&        |
| <code>sc[0...scptr-1]</code>                 | $\forall i \in \mathbf{BV}[32].$        |
| is decoded                                   | <code>i &lt; scptr</code> $\Rightarrow$ |
|                                              | <code>sc[i] == origsc[i]</code>         |

---

Then, after iteration:

---

|                               |                                                               |
|-------------------------------|---------------------------------------------------------------|
| <code>rC</code> decrements    | <code>rC<sub>After</sub> == rC-1</code>                       |
| <code>scptr</code> increments | <code>scptr<sub>After</sub> == scptr+1</code>                 |
| <code>sc[0...scptr]</code>    | $\forall i \in \mathbf{BV}[32].$                              |
| is decoded                    | <code>i &lt;= scptr</code> $\Rightarrow$                      |
|                               | <code>sc<sub>After</sub>[i] == origsc[i]</code>               |
| If <code>rC != 0</code>       | <code>eip<sub>After</sub> == rC<sub>After</sub> != 0 ?</code> |
| Loop again                    | <code>@loop_body :</code>                                     |
| Otherwise terminate           | <code>@done</code>                                            |

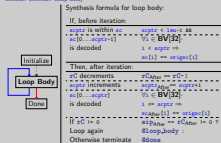


## Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

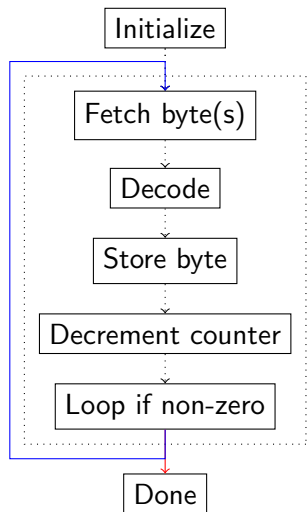


The loop body formula is more complicated. Remember, our loop invariants and variants consist of two parts: some statement about the machine state before the loop executes, and a statement about the machine state after the loop executes. So, before an iteration, if we haven't reached the end of the shellcode array, and if all of the shellcode bytes so far have been decrypted...

Then, after the iteration, the counter register has decremented, the shellcode pointer has incremented, an extra byte has been decoded, and `eip` is either the beginning of the loop again, or it's the address after the loop ends.

# 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.

## Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders



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.

With the construction that we just showed, we specified the behavior for the entire loop body. This means that the solver can generate solutions where the counter decrement is before or in the middle of the decoding. It gives the solver more freedom to come up with a solution, but it also makes the formula harder to solve. We can make it easier by breaking the loop's responsibilities down into parts, and specify how the individual parts are supposed to behave. This does exclude solutions where the parts are mixed together with one another.

# Synthesizing Decoders

## Synthesized Decoder: No Encoding Restrictions

```
F3 40 00 00 00 mov r3, 40h
@here:
E3 FF FF FF FF add r3, 0FFFFFFFh
C0 getscbyte r0
C8 putsctype r0
FB F7 jnz r3, @here
```

- ▶ Here is a decoder synthesized with no encoding restrictions.
  - ▶ Not very interesting.

└ Extensions

└ Synthesizing Decoders

└ Synthesizing Decoders

```
FS 40 00 00 00 mov r3, 40h
@shell:
E3 FF FF FF FF add r3, 0FFFFFFFh
C0 getsbyte r0
CB putsbyte r0
FB FF jnz r3, @shell
```

- Here is a decoder synthesized with no encoding restrictions.
- Not very interesting.

So here's a basic decoder that I synthesized without specifying any restrictions on the shellcode bytes. Because our loop iterations must increase the shellcode pointer, and the only way to do that is with the `putsbyte` instruction, the loop has to have a `putsbyte` instruction in it. Since `putsbyte` overwrites a byte of the shellcode, the code has to `getsbyte` to get the current value. The rest of the instructions relate to establishing the counter before the loop, decrementing it within the loop, and performing a conditional jump as the last instruction. It isn't very interesting, because I didn't specify any encoding constraints.

# 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, 0FFFFFFFh
D3 neg r3
CB putsctype r3
FB FD jnz r5, @here
```

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



## Synesthesia: A Modern Approach to Shellcode Generation

└ Extensions

└─ Synthesizing Decoders

└─ Synthesizing Decoders

## Synthesizing Decoders

Synthesized Decoder: Some Bytes Randomly Disallowed

```
FS 40 00 00 00 mov r5, 40h
@here:
C3 getbyte r3
E8 FF FF FF FF add r5, 0FFFFFFFh
D3 neg r3
CB putcbyte r3
FB FD jnz r5, @here
```

- Some bytes in the shellcode were randomly disallowed.

- Applying `neg` to each byte bypassed the restriction.

With this example, I took 64 random bytes, and then specified in the encoding constraint that some bytes were illegal. So this example does have to encode and decode the shellcode. We can see the `neg r3` instruction in the middle of the loop; it turns out that that was enough to bypass the restrictions that I put in place.

# Synthesizing Decoders

## Synthesized Decoder: Encode to Printable Characters

This example encodes each byte using two printable bytes.

```
mov r7, 40h
@here:
xor r5, 000000A4h
getcbyte r0
mov r2, 80008081h
getcbyte r1
add r1, r1
add r0, r1
putcbyte r0
add r7, 0FFFFFFFh
jnz r7, @here
```

Decoded/Encoded

|    |  |    |    |
|----|--|----|----|
| A0 |  | 50 | 28 |
| 8C |  | 4A | 21 |
| 1C |  | 32 | 75 |
| 29 |  | 3D | 76 |

NOP instructions are in red.

## Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

## Synthesizing Decoders

Synthesized Decoder: Encode to Printable Characters

This example encodes each byte using two printable bytes.

```

mov r7, 40h
Ghere:
mov r5, 0000044h
getabyte r0
mov r2, 8000801h
getabyte r1
add r1, r1
add r0, r1
putabyte r0
add r7, 0FFFFFFFh
jmp r7, Ghere

```

| Decoded/Encoded |       |
|-----------------|-------|
| 40              | 50 28 |
| 80              | 4A 21 |
| 1C              | 32 75 |
| 29              | 3D 76 |

NOP instructions are in red.

This example allows uses two encoded bytes to represent one decoded byte, where both encoded bytes are restricted to being printable. We can see two NOP instructions in red. The encoding method and decoder that it came up with are kind of unusual; I had to sort of reverse engineer it to figure out what it was doing. I leave that as an exercise for the reader.

# Synthesizing Decoders

Synthesized Decoder: Encode to ASCII Alphanumeric

This example encodes each byte using two alphanumeric bytes.

```
 mov r6, 40h
@here:
 getsb r4
 add r4, 43D087B6h
 mov r2, r4
 add r2, r2
 getsb r0
 xor r2, r0
 putsb r2
 add r6, 0FFFFFFFh
 jnz r6, @here
```

Decoded/Encoded

|    |  |    |    |
|----|--|----|----|
| A0 |  | 30 | 6C |
| 8C |  | 38 | 50 |
| 1C |  | ?? | ?? |
| 29 |  | ?? | ?? |

NOP instructions are in red.

## Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Synthesizing Decoders

Synthesizing Decoders

## Synthesizing Decoders

Synthesized Decoder: Encode to ASCII Alphanumeric

This example encodes each byte using two alphanumeric bytes.

```

mov r6, 40h
@here:
getachyte r4
add r4, 43D08780h
mov r2, r4
add r2, r2
getachyte r0
xor r2, r0
putachyte r2
add r0, 0FFFFFFFh
jmp r0, @here

```

Decoded/Encoded

|    |       |
|----|-------|
| 40 | 30 6C |
| 80 | 38 50 |
| 1C | 77 77 |
| 28 | 77 77 |

NOP instructions are in red.

This example allows uses two encoded bytes to represent one decoded byte, where both encoded bytes are restricted to being alphanumeric. We can see one NOP instructions in red – we don't actually need the register `r4` in this example. The encoding method and decoder that it came up with are really weird. They actually do not work for every possible input byte. There are 10 bytes that can't be encoded using this method – but none of those bytes were in my input, so it doesn't matter. Again, I leave it to the reader to figure out how this works.

## 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?

## Extensions

## Input State Preconditions

## Input State Preconditions

```
call [i+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?

Now, another extension of the idea. Sometimes, there is something specific that the shellcode needs to do, such as retrieve its own address in memory. It might be the case that none of the methods to do this can be encoded within a given encoding restriction.



# 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.

$$\begin{array}{c} \exists \overrightarrow{\text{COMPONENTS}} \\ \forall \overrightarrow{\text{INPUTS}} \\ \phi_{\text{Program}} \\ \overrightarrow{\text{INPUTS}}[\text{mem}[\text{esi}+4]] == \&\text{shellcode} \Rightarrow \phi_{\text{Functionality}} \quad \blacktriangleleft \end{array}$$

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

## Synesthesia: A Modern Approach to Shellcode Generation

Extensions

Input State Preconditions

Input State Preconditions

## Input State Preconditions

Conceptually

- What if, by virtue of our exploit scenario, we know that `[eax+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.

```

COMPONENTS
 INPUTS
 @Program
INPUT{[eax+4]== &shellcode} => @functionality

```

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

It might be the case that we know something special about the state of the machine at the time when our shellcode executes. For example, if we know that some register points to the shellcode, or that some memory location contains the address of the shellcode, that we can use that in place of a generic GETPC operation. We can take advantage of this information very naturally: essentially we just encode within the formula that the register or memory location contains the necessary address, and then any shellcode that we generate will be able to take advantage of that information automatically. So in this case, it gives us a way to implement GETPC that is specialized to our situation, whereas we otherwise would not be able to encode it.

## Extensions

Encoding Restrictions

Synthesis of Equivalent Snippets

Finding the Shortest Program

Synthesizing Decoders

Input State Preconditions

**Integration with Exploit Generation**

# Integration with Exploit Generation

```
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.

## Extensions

## Integration with Exploit Generation

## Integration with Exploit Generation

```
input = rax();
if (!validate(input)) return;
trace = transform(input);
vex_exec(trace);
```

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

One more extension before we finish up. Basically, automated exploit generation systems collect an execution trace from the point where the input enters the process, until the point where the vulnerability is triggered, and then it tries to generate an input that triggers the vulnerability. We can piggyback on existing automated exploit generation systems to also specify the behavior of the shellcode we wish to execute, to cause it to not only generate an input, but also shellcode. There are some limitations to this idea: basically, the constraints generated in this fashion are very rigidly tailored to an execution trace, so it may restrict shellcode generation more than what the program will actually tolerate.

## Discussion

Limitations

Evaluation

Future Work

Source Release

# Limitations

## Constraints Can Be Hard

```
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.

Discussion

Limitations

Limitations

```
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.

Now we'll discuss limitations of the idea. First, as with anything that uses an SMT solver, the queries can be hard to solve. On this slide, I am supposing that the output of some cryptographic hash function is used to trigger a vulnerability. For us to generate inputs that have specific properties, we need to invert the hash function. Obviously, that is very difficult, and we shouldn't expect to be able to solve those constraints in a reasonable amount of time.



# Limitations

## Can't Quantify Over Arrays

$\exists$  `mc[256]: char Array` · ◀  
...

- ▶ **Problem:** YICES won't let us quantify over arrays.

# Limitations

## Can't Quantify Over Arrays

```
∃ mc[256]: char Array · ◀
 ...
```

```
∃ mc00: char, mc01: char ... ◀
 ...
```

```
char get_byte(char idx) {
 switch(idx) {
 case 0x00: return mc00;
 case 0x01: return mc01;
 ...
```

- ▶ **Problem:** YICES won't let us quantify over arrays.
- ▶ **Solution:** Quantify over bytes; simulate array access.
  - ▶ Not a very serious limitation for synthesizing arrays.

## Synesthesia: A Modern Approach to Shellcode Generation

Discussion

Limitations

Limitations

## Limitations

Can't Quantify Over Arrays

```
∃ w:[256]: char Array ...
...
∃ w00: char, w01: char ...
...

char get_byte(char idx) {
 return idx;
}
case $w00: return w00;
case $w01: return w01;
...
```

- **Problem:** YICES won't let us quantify over arrays.
- **Solution:** Quantify over bytes; simulate array access.
  - Not a very serious limitation for synthesizing arrays.

Now for the second restriction, which is also a big deal. I used the SMT solver YICES to implement the prototype, since it has a special solver for the types of formulas that we generate. However, YICES doesn't let us use arrays in quantifiers. So throughout the presentation, I used "exists machine code array" to specify that our components in machine code synthesis were an array of bytes. However, it turns out that we can't actually do that, because we can't use arrays in quantifiers.

## Synesthesia: A Modern Approach to Shellcode Generation

Discussion

Limitations

Limitations

## Limitations

Can't Quantify Over Arrays

```
∃ w:[256]: char Array ...
...

∃ w00: char, w01: char ...
...


char get_byte(char idx) {
 return idx;
 case $w00: return w00;
 case $w01: return w01;
 ...
}
```

- **Problem:** YICES won't let us quantify over arrays.
- **Solution:** Quantify over bytes; simulate array access.
  - Not a very serious limitation for synthesizing arrays.

However, this is not a very serious problem. Instead, I can just have 256 bytes worth of machine code, and use the function `get_byte` to simulate array access during instruction encoding. This solves the problem with no serious effects.

# Limitations

## Can't Quantify Over Arrays

```
∃ mc[256]: char Array ·
∀ in: state · 
...
```

- ▶ **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.

## Synesthesia: A Modern Approach to Shellcode Generation

Discussion

Limitations

Limitations

## Limitations

Can't Quantify Over Arrays

```
∃ s:[256]: class Array *
 ∀ ic: state *
 ...
```

- **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.

But there is a second dimension to this limitation which is more serious. Since we can't quantify over arrays, we can't use arrays as part of our machine state. This is no big deal for the registers and flags, but it is a killer for memory accesses, since we use arrays to model memory. So my formulas can't synthesize any instructions that access memory. Note that this problem is not a mathematical problem, but rather, a shortcoming in the current implementation. It is possible that YICES will support this in the future; it's possible that Z3 or some other solver already supports it. In other words, while this is a major limitation, it's also one that can be overcome.

# 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.

Discussion

Limitations

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.

And on the theoretical side, one permanent limitation is that we will never be able to synthesize something that has an unspecified number of instructions in it – we can only synthesize programs that are a specific length, or “up to” a specific length. Also, when synthesizing decoders, my loop invariants and variants are very rigid, and discard a large number of possible decoder loops. These problems can be addressed theoretically – I’ve already begun working on them. Nevertheless, for now, only one loop schema is supported.



# Limitations

## Current Capabilities

| Capability               | SIMPLEMC       | X86            |
|--------------------------|----------------|----------------|
| Straight-line, No Memory | ✓              | ✓              |
| Straight-line, Memory    | X <sub>S</sub> | X <sub>S</sub> |
| Equivalent Snippets      | ✓              | ✓              |
| Shortest Solution        | ✓              | ✓              |
| Decoder Synthesis        | ✓              | X <sub>S</sub> |
| Input Preconditions      | ✓              | ✓              |
| Exploit Generation       | X <sub>P</sub> | X <sub>E</sub> |

Legend:

- ▶ ✓: support is present.
- ▶ X<sub>S</sub>: not supported due to solver limitations.
- ▶ X<sub>E</sub>: not supported due to external requirements.
- ▶ X<sub>P</sub>: not supported due to pointlessness.

## Synesthesia: A Modern Approach to Shellcode Generation

Discussion

Limitations

Limitations

Limitations  
Current Capabilities

| Capability               | SIMPLEC        | X86            |
|--------------------------|----------------|----------------|
| Straight-line, No Memory | ✓              | ✓              |
| Straight-line, Memory    | X <sub>S</sub> | X <sub>C</sub> |
| Equivalent Snippets      | ✓              | ✓              |
| Shortest Solution        | ✓              | ✓              |
| Decoder Synthesis        | ✓              | X <sub>S</sub> |
| Input Preconditions      | ✓              | ✓              |
| Exploit Generation       | X <sub>S</sub> | X <sub>C</sub> |

Legend:

- ✓: support is present.
- X<sub>S</sub>: not supported due to solver limitations.
- X<sub>C</sub>: not supported due to external requirements.
- X<sub>0</sub>: not supported due to pointlessness.

So here's a summary of the current implementations and their limitations. We can synthesize straight-line programs without memory accesses for both SIMPLE and X86. We can synthesize equivalent programs for both of those languages, and find the shortest solutions. Decoder synthesis only works for SIMPLE, since there was no point in modifying the X86 instruction set for the sake of a contrived implementation. We can deal with input preconditions for both languages. Integration with automated exploit generation is not supported, as that relies on external tools.

# Evaluation

| Task                                         | Time (s) |
|----------------------------------------------|----------|
| SIMPLEMC 2-line <code>r0 = 0</code>          | 0.0      |
| SIMPLEMC shortest 2-line <code>r0 = 0</code> | 6        |
| SIMPLEMC longest 2-line <code>r0 = 0</code>  | 0.0      |
| SIMPLEMC empty decoder                       | 127      |
| SIMPLEMC exclude bytes decoder               | 153      |
| X86 3-line <code>eax = 12345678h</code>      | 0.6      |

Discussion

Evaluation

Evaluation

| Task                                      | Time (s) |
|-------------------------------------------|----------|
| SIMPLEMC 2-line $\forall 0 = 0$           | 0.0      |
| SIMPLEMC shortened 2-line $\forall 0 = 0$ | 0        |
| SIMPLEMC longest 2-line $\forall 0 = 0$   | 0.0      |
| SIMPLEMC empty decoder                    | 127      |
| SIMPLEMC exclude bytes decoder            | 153      |
| X86 3-line $\text{eax} = 12345678h$       | 0.6      |

I tried to be very clear throughout the presentation that this can be slow. If you weren't convinced then, you should be convinced now. Basically, small sequences of code with loose restrictions can usually be generated quickly. When the sequences get large, or the restrictions become onerous, Synesthesia begins to take a long time and consume a lot of memory.

# 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?

Discussion

Future Work

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?

Here are some ideas for making things faster. First, stochastic superoptimization is a program synthesis technique based on genetic algorithms. I think it would be particularly suitable in this case. Perhaps you could integrate some sort of grammar-based mutation framework based upon a specification of the encoding restrictions. Next, we could try to specialize our SMT solver for this particular problem. It might make sense to profile YICES as it's running to see if there are any obvious inefficiencies in its search procedure. The EXISTS/FORALL solver uses a special algorithm based on generalizing implicants; that could maybe be specialized to this particular problem. And finally, maybe there are improvements to be gained from explicitly creating a SMT theory for X86 instruction synthesis.

# Discussion

## Source Release

- ▶ 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.

2016-11-08

# Synesthesia: A Modern Approach to Shellcode Generation

Discussion

Source Release

Discussion

Discussion  
Source Release

- Source release includes many YICES scripts that demonstrate the features shown in this presentation.
  - Both X86 and the SIMPLIMC language.
- Soon I'll announce the URL on my twitter account, @R011R011s.

I haven't uploaded the code to the Internet just yet. I'm going to do that once I get back and have a chance to clean it up again. Follow me on twitter for the URL, which I'll tweet in the near future.



# 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<sup>1</sup>.
- ▶ Available now for private offerings!
- ▶ **See website for public classes (January, Maryland, USA).**

---

<sup>1</sup>ROP compiler application subject to potential replacement pending forthcoming regulation of the computer security industry

└ 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 C3 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  $\rightarrow$  IR translator, ROP compiler<sup>1</sup>.
- Available now for private offerings!
- See [website for public classes \(January, Maryland, USA\)](#).

<sup>1</sup>ROP compiler application subject to potential replacement pending forthcoming regulation of the computer security industry

If you liked my presentation, or want to learn more about SMT solvers, I teach training classes on this subject. I don't assume you know much about math or academic computer science. Basically we code a complete binary analysis platform in Python and use it to approach reverse engineering problems. The class is available publicly – there is an offering in January in Maryland – and also privately. Please see my website for more details.

# 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.

└─ Conclusion

└─ 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.

So that's it. This is to my knowledge the first approach at generic shellcode construction under arbitrary restrictions. Right now it's works alright for smaller sequences with simpler encodings, but it's still too slow for some of the crazier extensions I discussed in the presentation. Feel free to take a look at the code and play with it. And now I'll take any questions.