Understanding Symbolic Execution

Another step forward to the day when computers do everything humans do but better.
A Simple Capture-the-Flag Level

# A simple guessing game.
user_input = raw_input('Enter the password: ')
if user_input == 'hunter2':
    print 'Success.'
else:
    print 'Try again.'

There are many solutions:
  • Use objdump or readelf to find the string ‘hunter2’.
  • Use ltrace to find the comparison.
  • Use a debugger and inspect the memory where the password is stored.
  • ...
A Complex Capture-the-Flag Level

# A complex guessing game. Don’t bother to figure out what the code does.
def encrypt(string, amount):
    for i in range(0, len(string)):
        string[i] += amount

user_input = raw_input('Enter the password: ')
if encrypt(user_input, amount=1) == encrypt('hunter2', amount=2):
    print 'Success.'
else:
    print 'Try again.'

There are many not as many solutions:
• Use objdump or readelf to find the string ‘hunter2’. No, ‘hunter2’ isn’t the password anymore.
• Use ltrace to find the comparison. No, the strings it compares aren’t what you entered nor the password.
• Use a debugger and inspect the memory where the password is stored. Nope. Same as above.
• Reverse engineer the encrypt function. Simple in this example, but it could be really complex in a different one.
Solution: **Symbolic Execution**

What’s that? **It’s a system that walks through all possible paths of a program.**

Let’s work through an example.

```python
1 user_input = raw_input('Enter the password: ')
2 if user_input == 'hunter2':
3   print 'Success.'
4 else:
5   print 'Try again.'
```

We want to find an input that arrives here →

*some: for many programs, this would be impossible (even if we decided that a path that never halts is considered the “end” of a branch, the halting problem shows that determining when a branch is considered done is undecidable), and for others it would take longer than the universe has existed so far to traverse all paths.*
Step 1: **Inject a Symbol**

```python
user_input = raw_input('Enter the password: ')
if user_input == 'hunter2':
    print 'Success.'
else:
    print 'Try again.'
```

But first, what is a **symbol**?
What is a Symbol?

\[ x^2 + 2x + 3 = 4 \]

Remember high school algebra?

Think of a symbol as \( x \), except that it’s a variable in the program. We don’t know what \( x \) is. We want to find out. Same with a symbol.

\( x \) depends on the equation(s) that constrain it.

A symbol depends on the execution path(s) that constrain it.

…but wait, what is an execution path?
What is an Execution Path?

It’s a possible way to travel through the program.

For example...

```python
1 user_input = raw_input('Enter the password: ')
2 if user_input == 'hunter2':
3     print 'Success.'
4 else:
5     print 'Try again.'
```

...has two possible execution paths. Can you see them?
Execution Path Example

```python
1 user_input = raw_input('Enter the password: ')
2 if user_input == 'hunter2':
3   print 'Success.'
4 else:
5   print 'Try again.'
```

Path 1: if `user_input` equals 'hunter2'

Path 2: if `user_input` does not equal 'hunter2'

...okay that makes sense. But how do execution paths act like equations that constrain symbols?
How do execution paths constrain symbols?

```
user_input = λ
if user_input == 'hunter2'
print 'Success.'
```

Look familiar? It’s the same as on the last slide, but `user_input` is now a symbol.

For this path to be executed, the symbol, λ, must be equal to ‘hunter2’.

Otherwise, the computer would execute the other path.
Return to Step 1: Inject a Symbol

You are here

1 user_input = λ
2 if user_input == 'hunter2':
3   print 'Success.'
4 else:
5   print 'Try again.'

Goal: Find the execution path that reaches line 3, then solve for λ.
Step 2: Branch

1 user_input = λ
2 if user_input == 'hunter2':
3    print 'Success.'
4 else:
5    print 'Try again.'

What happens when you reach an if statement that depends on a symbol? You branch.

...please explain!
What does it mean to **branch**?

**Branching** just means to split into the **different possible execution paths**.

```python
1 user_input = λ
2 if user_input == 'hunter2':
3   print 'Success.'
4 else:
5   print 'Try again.'
```

Path 1: if `user_input` **equals** 'hunter2'

Path 2: if `user_input` **does not equal** 'hunter2'

```python
user_input = raw_input('Enter the password: ')
if user_input == 'hunter2'  # it is equal!
    print 'Success.'
else:
    print 'Try again.'
```

...isn’t this is the same slide as before, copied and pasted?
Step 3: Evaluate each Branch

Let’s imagine we picked the ‘user_input does not equal “hunter2”’ branch first.

```python
1 user_input = λ
2 if user_input == 'hunter2':
3     print 'Success.'
4 else:
5     print 'Try again.'
```

We are at the end of the execution and didn’t find what we wanted. Continue with the other branch!
Step 3: Evaluate each Branch (part 2)

Now we chose the ‘user_input equals “hunter2”’ branch.

```
1 user_input = λ
2 if user_input == 'hunter2':
3     print 'Success.'
4 else:
5     print 'Try again.'
```

We found what we wanted! We now have an execution path that can constrain the symbol. We can solve for $\lambda$ to find the password.
A More Complex Example: Part 1

The following is source code from Ch06CAsm_Conditionals:

```c
#define SECRET 100
int check_code(int input) {
    if (input >= SECRET+88) return 0;
    if (input > SECRET+100) return 0;
    if (input == SECRET+68) return 0;
    if (input < SECRET) return 0;
    if (input <= SECRET+78) return 0;
    if (input & 0x1) return 0;
    if (input & 0x2) return 0;
    if (input & 0x4) return 0;
    return 1;
}
```

What are the possible paths?
#define SECRET 100

int check_code(int input) {
    if (input >= SECRET + 88) return 0;
    if (input > SECRET + 100) return 0;
    if (input == SECRET + 68) return 0;
    if (input < SECRET) return 0;
    if (input <= SECRET + 78) return 0;
    if (input & 0x1) return 0;
    if (input & 0x2) return 0;
    if (input & 0x4) return 0;
    return 1;
}

Paths can be represented as a tree.
We can perform any **tree search algorithm** to find the node that **returns 1**.

**Breadth-first search** is a great choice (and, by default, what Angr uses.)

Why not use **Depth-first search**?
Solving a More Complex Example: Part 2

Once we have a path, we can build an equation that can be solved with a satisfiability modulo theories (SMT) solver:

\[
\text{input} \geq \text{SECRET} + 88 \\
\land \text{input} > \text{SECRET} + 100 \\
\land \text{input} = \text{SECRET} + 68 \\
\land \text{input} < \text{SECRET} \\
\land \text{input} \leq \text{SECRET} + 78 \\
\land \text{input} \& \ 0x1 \\
\land \text{input} \leq \text{SECRET} + 78 \\
\land \text{input} \& \ 0x2 \\
\land \text{input} \& \ 0x4
\]

Remember, SECRET = 100.
The Real World™

user_input0 = \lambda_0
user_input1 = \lambda_1
user_input2 = \lambda_2

... if user_input0 == 'hunter2':
    if 2 * user_input1 - 7 * user_input2 < len(user_input0):
        ... # more complex functionality
        print 'Success.'
    else:
        print 'Try again.'
else:
    print 'Try again.'

Of course, in the real world, the binaries will be complex. There could be many symbols and many branches. The exponential growth of the complexity of the binary is symbolic execution’s largest problem.
How do we step through the program, find the branch we want, and solve for $\lambda$?

We let the computer do that! Our friends at UCSB built a powerful tool called Angr to do that for us. It’s written in Python and it operates on native binaries (no source code required!).

https://github.com/angr
Symbolic Execution CTF: Part 1

An Introduction to Path Groups
What is Angr?

Angr is a symbolic execution engine*. It can:

• Step through binaries (and follow any branch)
• Search for a program state that meets a given criteria
• Solve for symbolic variables given path (and other) constraints

*and more, but use of the included binary analysis tools unrelated to symbolic execution is out of the scope of these slides and the associated CTF.
Recall, the foundation of symbolic execution involves two principles:

1. Execution Paths
2. Symbols

We will begin by discussing execution paths.
An Execution Path in Angr

An execution path represents a possible execution of the program that begins somewhere and ends somewhere else.

A node of an execution path in Angr is represented by a ‘SimState’ object. As the name suggests, it stores the state of the program, as well as a history of the previous states. Chaining these SimStates together creates a path.
A single execution path isn’t interesting. We can view one by running the program with a given input without Angr.

Instead, we care about all* (as many as possible) execution paths, so that we can search them to find the one we want.

We’ll talk about searching later, but first, how do we represent a set of execution paths in Angr, and how do we build them?
A Simulation Manager (simgr) in Angr

Angr stores and handles a set of possible paths for a given program in a ‘simulation manager’ object.

Simulation managers provide functionality to step through the program to generate possible paths/states.
Building a Set of Paths

1. Angr **starts the program** wherever you instruct it to start (this is the first active state)

2. Execute instructions in each active (nonterminated) state until we reach a branching point or the state terminates

3. At every branching point, **split the state** into multiple states, and add them to the set of active states

4. Repeat step 2..4 until we find what we want or all states terminate
Animation: Building a Set of Paths

if (input >= SECRET+88)

Legend:
Blue = already executed
Green = active
Red = terminated
Animation: Building a Set of Paths

if (input >= SECRET+88)
    return 0;

if (input > SECRET+100)

Legend:
Blue = already executed
Green = active
Red = terminated
Animation: Building a Set of Paths

```plaintext
if (input >= SECRET+88)
    return 0;

if (input > SECRET+100)
    return 0;

if (input == SECRET+68)
    return 0;
```

Legend:
Blue = already executed
Green = active
Red = terminated
if (input >= SECRET+88)
    return 0;

if (input > SECRET+100)
    return 0;

if (input == SECRET+68)
    return 0;

if (input < SECRET)
    return 0;

Legend:
Blue = already executed
Green = active
Red = terminated
if (input >= SECRET + 88)
    return 0;
    if (input > SECRET + 100)
        return 0;
        if (input == SECRET + 68)
            return 0;
            if (input < SECRET)
                return 0;
                if (input <= SECRET + 78)
                    return 0;
                    return 0;

Legend:
Blue = already executed
Green = active
Red = terminated
if (input >= SECRET + 88)
    return 0;

if (input > SECRET + 100)
    return 0;

if (input == SECRET + 68)
    return 0;

if (input < SECRET)
    if (input <= SECRET + 78)
        return 0;

... etc

Legend:
Blue = already executed
Green = active
Red = terminated
if (input >= SECRET + 88)
  return 0;
if (input > SECRET + 100)
  return 0;
if (input == SECRET + 68)
  return 0;
if (input < SECRET)
  return 0;
if (input <= SECRET + 78)
  return 0;
if (input & 0x1)
  return 0;
if (input & 0x2)
  return 0;
if (input & 0x4)
  return 0;
return 1;

We found what we wanted!

Legend:
Blue = already executed
Green = active
Red = terminated
Searching for What We Want

<table>
<thead>
<tr>
<th>Method 1: Search for an instruction address</th>
<th>Method 2: Search for anything else!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perhaps we want to find this address.</td>
<td>Perhaps we want to find when the variable ‘success’ is equal to true.</td>
</tr>
<tr>
<td>804867a: sub $0xc,%esp</td>
<td>Any arbitrary function that determines if we have reached a state we want would work.</td>
</tr>
<tr>
<td>804867d: push $0x8048760</td>
<td></td>
</tr>
<tr>
<td>8048682: call 8048400 <a href="mailto:backdoor@plt">backdoor@plt</a></td>
<td></td>
</tr>
<tr>
<td>8048687: add $0x10,%esp</td>
<td></td>
</tr>
</tbody>
</table>

Both of these approaches are trivial! At each step, just check if any active state meets your conditions.
One of the biggest problems with symbolic execution:

With each if statement, the number of possible branches might double. The growth of the problem is exponential with respect to the size of the program.

There is no known good solution to this problem.
One Good Approach: Avoiding States

If you can identify conditions that would indicate that it is unlikely that continuing would lead to a successful state, you can terminate the path immediately and cross off large sections of the state graph.

Right here a variable is set that tells us that we will not find anything useful down this path.

We only have to search half of the graph! Saves a lot of time.

The successful state is here.
Avoiding Paths

We can avoid states in the exact same way that we accept them as successful.
How do we determine which conditions might lead to a failed state?

Human intuition!

Also, there are various heuristic algorithms that are mostly out of the scope of these notes. We will briefly touch on a method called Veritesting much later.

If you find a better way, publish it!
Summary: Algorithm for Find and Avoid

• Load the binary

• Specify a starting point and create a simulation manager

• While we have not found what we want...
  • Step all active states
  • Run our ‘should_accept_state’ predicate on each active state
    • If one accepts, we found what we wanted! Exit the loop
  • Run our ‘should_avoid_state’ predicate on each active state
    • For each state that is accepted, mark it for termination
  • Remove all state that are marked for termination from the set of active states
Shortcut: The ‘Explore’ Method

The previous algorithm is so common that Angr wrote a single function to do it for you, called the ‘explore’ function:

```
simulation.explore(find=should_accept_path, avoid=should_avoid_path)
```

... will add any path that is accepted to the list ‘simulation.found’

Additionally, searching or avoiding a specific instruction address is common enough that the find and avoid parameters also accept addresses:

```
simulation.explore(find=0x80430a, avoid=0x9aa442)
```

... would search for address 0x80430a and terminate anything that reaches 0x9aa442.
Symbolic Execution CTF: Part 2

Introducing Symbols and Constraints
Injecting Symbols

In some cases, Angr automatically injects a symbols when user input is queried from the stdin file. *

When Angr does not automatically inject a symbol where we want one, we can do so manually.

* It does this with what are called SimProcedures, which we will cover later.
Angr’s symbols are represented by what it calls bitvectors. Bitvectors have a size, the number of bits they represent. As with all data in programming, bitvectors can represent any type that can fit. Most commonly, they represent either n-bit integers or strings.

The difference between a bitvector and a typical variable is that, while typical variables store a single value, bitvectors store every value that meet certain constraints.
Let the bitvector $\lambda$ be 8 bits and be constrained by:
\[
( \lambda > 0, \quad \lambda \leq 4, \quad \lambda \mod 2 = 0 ) \lor ( \lambda = 1 )
\]

The above is how the bitvector would be stored.

However, if we were to concretize the bitvector (collapse it to a specific value), it could take on any of the following values: 2, 4, or 1.
Between Symbolic and Concrete

Definitions:

A **concrete** bitvector: a bitvector that can take on exactly 1 value.
(Example: \{ \lambda: \lambda = 1 \})

A **symbolic** bitvector: a bitvector that can take on more than 1 value.
(Example: \{ \lambda: \lambda > 10 \})

An **unsatisfiable** bitvector: a bitvector that cannot take on any values.
(Example: \{ \lambda: \lambda = 10, \lambda \neq 10 \})

An **unconstrained** bitvector: a bitvector that can take on any value, within the bounds of its size.
Angr provides a nice frontend to z3, an open-source constraint solver. It has the following functionality (and more):

- Find any (single) value of a bitvector
- Find up to $n$ possible values of a bitvector
- Find the maximum or minimum possible values of a bitvector
- Determine if a bitvector is ‘true’ or ‘false’
- Determine if a bitvector is satisfiable
Symbols in the Context of a Program State

```
1 user_input = λ
2 if user_input == 'hunter2':
3    print 'Success.'
4 else:
5    print 'Try again.'
```

We can inject symbols into variables as long as the size of the bitvector is equal to the size of the variable.

Constraints are automatically generated (ex: $\lambda = \text{‘hunter2’}$, or, for the other path, $\lambda \neq \text{‘hunter2’}$) as the engine follows a given path.

If we desire, we can manually add constraints to any bitvector at any time during the execution of the program.
Symbol Propagation

user_input = \lambda
encrypted_input0 = user_input - 3
encrypted_input1 = encrypted_input0 + 15
encrypted_input2 = encrypted_input1 * 7

To the right you see the memory, with the variables user_input and encrypted_inputX marked.

All are symbolic, represented by the green.

The variables encrypted_inputX depend entirely on user_input, denoted by the arrow.

Symbols can propagate when values are transferred.
Constraint Propagation

user_input = \lambda
encrypted_input0 = user_input - 3
encrypted_input1 = encrypted_input0 + 15
encrypted_input2 = encrypted_input1 \times 7

In this example, if we add the constraint: \lambda = 10, then:

user_input = \lambda = 10
encrypted_input0 = user_input - 3 = 10 - 3 = 7
encrypted_input1 = encrypted_input0 + 15 = 7 + 15 = 22
encrypted_input2 = encrypted_input1 \times 7 = 22 \times 7 = 154

Constraints propagate through the program.
Constraint Reverse-Propagation

We can add constraints to propagated symbolic values!

\[
\begin{align*}
\text{user_input} &= \lambda \\
\text{encrypted_input0} &= \text{user_input} - 3 \\
\text{encrypted_input1} &= \text{encrypted_input0} + 15 \\
\text{encrypted_input2} &= \text{encrypted_input1} \times 7
\end{align*}
\]

In this example, if we add the constraint: \( \text{encrypted_input2} = 14 \), then:

\[
\begin{align*}
\lambda &= \text{user_input}, \quad \lambda = -10 \\
\text{user_input} - 3 &= \text{encrypted_input0} = -13, \quad \text{user_input} = -10 \\
\text{encrypted_input0} + 15 &= \text{encrypted_input1} = 2, \quad \text{encrypted_input0} = -13 \\
\text{encrypted_input1} \times 7 &= \text{encrypted_input2} = 14, \quad \text{encrypted_input1} = 2
\end{align*}
\]

Constraints propagate backwards through the program. We can solve for initial conditions given our desired results in this way.
A second huge problem with symbolic execution:

It relies on solving complex systems of constraints.

The constraint-satisfaction problem is NP-complete.

Need we say more?
Why are the first three levels of the CTF solved without injecting symbols?

Angr injects them for you!

Angr handles simple calls of ‘scanf’ (without complex format strings.)

You will need to inject symbols manually if the input is more complex, for example:

- Complex format string for scanf
  - From a file
  - From the network
  - From interactions with a UI

The following slides have a few examples of common patterns.
For simple cases, Angr replaces this for us, so that the user input function injects symbolic values into the registers.

For more complex cases, we need to inject the symbols ourselves. We start the program after the user input and initialize the registers with a symbolic value.
Injecting Symbols Example: Registers

Because the input is symbolic, the output will be symbolic; we can solve for the input given the output we want (just like always).
Injecting Symbols Example: Registers

Situation: The get_user_input function returns values by writing them to registers.
Solution: Instead of calling get_user_input, write symbolic values to the registers.

Start immediately after the call, here.

```
call 80487a5 <get_user_input>
mov %eax,-0xc(%ebp)
mov %ebx,-0x8(%ebp)
mov %ecx,-0x4(%ebp)
```

Evidence that the get_user_input function returned the user input on the registers.

In Angr, you can write to a register with either a concrete or a symbolic value:
```
state.regs.eax = my_bitvector
```
will write the value of my_bitvector to eax.
Injecting Symbols Example: Global Memory

User input writes to fixed points in memory.
Injecting Symbols Example: Global Memory

The same principle can be applied to global memory!
Injecting Symbols Example: Global Memory

Situation: The get_user_input function returns values by writing them to addresses determined at compile time.

Solution: Instead of calling get_user_input, write symbolic values to the addresses.

```
push $0xaf84128
push $0xaf84120
push $0x8048733
call 8048400 <scanf@plt>
```

Format string "%u %u"

Parameters (scanf will write user input to these addresses)

scanf("%u %u", &a, &b)

In Angr, you can write to an address with either a concrete or a symbolic value:

```
state.memory.store(0xaf84120, my_bitvector)
```

will write the value of my_bitvector to 0xaf84120.
Injecting Symbols Example: The Stack

... and, of course, the stack!
Injecting Symbols Example: The Stack

What about the stack?

Allocate memory for local variables
- `sub $0x20, %esp`
- `lea -0x8(%ebp), %eax`
- `push %eax`
- `push $0x80489c3`
- `call 8048370 <scanf@plt>`

State variables

Specify a specific local variable as a parameter to scanf

In Angr, you can push to the stack with either a concrete or a symbolic value:
- `state.stack_push(my_bitvector)`
  - will push the value of my_bitvector to the top of the stack.

You may need to account for anything you don’t care about at the beginning of the stack by adding padding:
- `state.regs.esp -= 4`
  - adds 4 bytes of padding.
Injecting Symbols Example: Dynamic Memory

- Memory allocated on the heap (potentially different every execution; unknown at the start)

- A pointer pointing to somewhere unknown.
Injecting Symbols Example: Dynamic Memory

We can overwrite this pointer to point to any address we want.

Memory we decided to use as symbolic memory

A dummy address we chose that nothing else uses.
Injecting Symbols Example: Dynamic Memory

What if you don’t know the memory location scanf to which scanf writes?

Scanf will write to the address stored in the pointer located at 0xaf84dd8

```
mov 0xaf84dd8,%edx
push %edx
push $0x8048843
call 8048460 <scanf@plt>
```

If you cannot determine the address to which scanf writes because it is stored in a pointer, you can overwrite the value of the pointer to point to an unused location of your choice (in this example, 0x4444444):

```
state.memory.store(0xaf84dd8, 0x4444444)
state.memory.store(0x4444444, my_bitvector)
```

At this point, the pointer at 0xaf84dd8 will point to 0x4444444, which will store your bitvector.
Injecting Symbols Example: The Filesystem

What if our user input function queries from the filesystem (or any other Linux file, including the network, the output of another program, /dev/urandom, etc)?
A program has access to an address space, which it uses to store instructions, the stack, the heap, and static data.

We have used it in Angr using the following functions:

- `state.memory.store(...)`
- `state.memory.load(...)`

We can use the same Python type as `state.memory` (which is `SimMemory`) to store other types of data, such as the contents of files!
Injecting Symbols Example: The Filesystem

File memory (/tmp/hello.txt)

We can make the file memory entirely symbolic.

We also have to give it a filename.

Note: our file memory is separate from our program memory. Address 0x0 in our file does not correspond to address 0x0 in our program.
Injecting Symbols Example: The Filesystem

In short: Angr allows you to specify an alternate, symbolic filesystem of your own specification. More information on this is included in the CTF.
Angr Implementation of Previous Examples

The implementation details are included with the CTF, in scaffoldXX.py, for the challenges that involve injecting symbolic memory and constraining it.
Symbolic Execution CTF: Part 3

Handling Non-trivial Behavior
Motivation: A Simple Example

The program iterates through 16 elements, each time it branches.

By the end of the loop, there will be a total of $2^{16}$ or 65,536 branches.

Could be reduced to:
user_input == 'Z' * 16
One branch.

```python
def check_all_Z(user_input):
    num_Z = 0
    for i in range(0, 16):
        if user_input[i] == 'Z':
            num_Z += 1
        else:
            pass
    return num_Z == 16
```
Solution

Of course, there are powerful algorithms to deduce the insight on the previous slide.

None work as well as human intuition for many cases (yet!).

For complex functions that can be easily simplified by a human, we can use Angr to replace the code with its summary, written in Python.
Hooks

We want to skip these instructions and instead run our own code.

You can do this using a hook. You specify an address to ‘hook’, the number of bytes of instructions you want to skip, and a Python function that will be run to replace the skipped instructions.

Note: the number of instructions you skip can be zero.
Hook Walkthrough

Let’s imagine we want to replace the call to `check_all_Z`, with our own check function:

```python
def replacement_check_all_Z():
    eax = (*0x804a420 == 'ZZZZZZZZZZZZZZ')
```

Return values are stored in `eax`
Hook Walkthrough

...  
8048774: add %edx,%eax  
8048776: sub $0x4,%esp  
804877c: push $0x804a420  
8048781: call 8048460 <check_all_Z>  
8048786: add $0x10,%esp  
8048789: test %eax,%eax  
804878b: jne 8048794 <main+0x19f>  
...

def replacement_check_all_Z():
    eax = (*0x804a420 == 'ZZZZZZZZZZZZZZZZ')

Call: binary.hook(0x8048776, length=16, replacement_check_all_Z)

Address we want to hook

The instructions are represented with 16 bytes in memory. If we didn’t want to skip any instructions (and run our Python code in addition to the instructions), we could let length=0.

Function to replace run when we reach our hook

(Syntax altered for rhetorical purposes, see CTF for actual syntax)
Hook Walkthrough

Call: `binary.hook(0x8048776, length=16, replacement_check_all_Z)`

19 bytes

```
8048774: add %edx,%eax
8048776: sub $0x4,%esp
804877c: push $0x804a420
8048781: call 8048460 <check_all_Z>
8048786: add $0x10,%esp
8048789: test %eax,%eax
804878b: jne 8048794 <main+0x19f>
```

\[ eax = (\ast 0x804a420 == 'ZZZZZZZZZZZZZZZ') \]

(Syntax altered for rhetorical purposes, see CTF for actual syntax)
Common Patterns

Hooks are useful for:

• Injecting symbolic values partway through the execution.

• Replacing complex functions.

• Replacing unsupported instructions (for example, most syscalls).
Complex Functions

Replacing complex functions with hooks is so common that Angr included sugar to make it easier.

A SimProcedure provides a simple way to replace a function with a summary in a Pythonic way.
Review of Functions

1. Push parameters to the stack
2. Push return address to the stack
3. Jump to function address
4. Handle parameters*
5. Execute function
6. Write return value to appropriate location
7. Pop return address and jump to it
8. Pop parameters

* The standard calling convention for programs compiled with gcc targeting IA-32 does not need to do anything with parameters once the function is called, since they are already on the stack, but you could imagine that a different calling convention might require the function to copy the parameter from, say, a register, onto the stack.
SimProcedure Algorithm

1. Push parameters to the stack
2. Push return address to the stack
3. Jump to function address
4. Handle parameters
5. Execute function
6. Write return value to appropriate location
7. Pop return address and jump to it
8. Pop parameters

Hooks here, at the beginning of the function address

Done automatically

Skips all instructions until the function is about to return, and resumes execution here

Allows user to replace this in a Pythonic way
SimProcedure Example

Ugly (hooks):

```python
def replacement_check_all_Z():
eax = (*0x804a420 == 'ZZZZZZZZZZZZZZZZ')
```

- Manually set return value
- Handle parameters ourselves

Beautiful (sim procedures):

```python
def replacement_check_all_Z(input):
    return input == 'ZZZZZZZZZZZZZZZZ'
```

- Uses Python's return function arguments
- Uses Python's return functionality

```
... 8048774: add %edx,%eax
  8048776: sub $0x4,%esp
  804877c: push $0x804a420
  8048781: call 8048460 <check_all_Z>
  8048786: add $0x10,%esp
  8048789: test %eax,%eax
  804878b: jne 8048794 <main+0x19f>
...
```

(Syntax altered for rhetorical purposes, see CTF for actual syntax)
SimProcedure in Practice

SimProcedures are used to replace anything you fully understand and don’t want to test for bugs, or that is unsupported by Angr.

Because the problem complexity scales exponentially with the length of the program, any and every function that meets the above criteria should be replaced with a SimProcedure, to save time.

Currently, the reimplementation of a (quickly expanding) subset of libc is included with Angr.
If SimProcedures are impractical...

... for example, if you do not understand a function, need to test it for bugs, or do not want to invest time to reimplement it ...

• Find a simpler version of the function(s)—glibc is complex, but an embedded version of libc might be simpler.

• Use an algorithm to automatically attempt to simplify the function
Veritesting and Why Human Intuition Always Wins

The Veritesting algorithm, developed at CMU, attempts to reduce state explosion by combining branches. It can automatically reduce:

```
def check_all_Z(user_input):
    num_Z = 0
    for i in range(0, 16):
        if user_input[i] == 'Z':
            num_Z += 1
        else:
            pass
    return num_Z == 16
```

Due to the difficult nature of reducing the algorithm, Veritesting relies on a heuristic to best determine how to merge states.

Of course, with current technology, humans still do this better.
For more information on Veritesting...


Statically-Linked Binaries

Instead of...

... Statically-linked binaries will look like this:

In order to replace libc functions with their corresponding already-implemented SimProcedures, Angr looks for the symbols ‘scanf@plt’ and ‘rand@plt’. If the program is statically-linked, it does not know what to replace with a SimProcedure.
A Solution to Statically-Linked Binaries

Binary

```c
scanf ...
rand ...
my_function ...
main ...
```

Solution: 
Manually hook them, as you have done already!

Implementation details are included with the CTF.
Analyzing Dynamic Libraries (and other binary formats)

We can begin wherever we want in the executable, in the same way as we have been doing in the CTF, using blank_state.

For position-independent code, we may need to specify a base address for the address space.

As always, implementation details are in the CTF.
Symbolic Execution CTF: Part 4

(An Intro to) Automatic Exploit Generation
1. Determine the type of exploit you want to search for, for example:
   - Arbitrary read (crash the program, read a password, etc)
   - Arbitrary write (inject shellcode, overwrite return address, etc)
   - Arbitrary jump (jump to your shellcode, return oriented programming, etc)
2. Write a Python function using Angr to determine if we have reached the condition necessary for the exploit.
3. Constrain the system in a way that would set up the attack.
4. Allow Angr to solve for the input that meets the constraints.
Simple Example: Arbitrary Read

An innocuous string

Buffer; stores input from stdin

‘puts’ prints the string this pointer points at

If we can find a buffer overflow that overwrites this pointer, we can print out the admin password

The admin password
Simple Example: Arbitrary Read

Buffer; stores input from stdin

The admin password

An innocuous string

Inject as much symbolic data as possible into the buffer! (In this case, the buffer is larger than the amount of input the program accepts; it is not vulnerable to attack.)

‘puts’ prints the string this pointer points at

Buffer; stores input from stdin
Simple Example: Arbitrary Read

Buffer; stores input from stdin

The admin password

An innocuous string

‘puts’ prints the string this pointer points at

Buffer; stores input from stdin

However, eventually we may come across an example that allows us to overwrite the pointer.
Simple Example: Arbitrary Read

Buffer; stores input from stdin

The admin password

An innocuous string

‘puts’ prints the string this pointer points at

Buffer; stores input from stdin

The question is: do we control it in such a way that we can point it at the admin password?

We determine this by constraining the pointer and letting z3 do the heavy lifting.
Simple Arbitrary Read Strategy

• Initialize state
• While we have not found a solution or exhaustively searched the binary:
  • For each active state:
    • If the program is calling puts (or printf, or send over the network, etc):
      • If the parameter (a pointer to a string to print) is symbolic and can be constrained to point to the memory address we want to read:
        • Constrain it as such
        • Solve for the user input
      • Step the active states
Simple Example: Arbitrary Write

If we can find a buffer overflow that overwrites this pointer, we can overwrite the admin password to be our own password!

Buffer; stores input from stdin

\textbf{ncpy} copies the data from the user to the buffer the pointer points to

The admin password

An innocuous buffer
Simple Example: Arbitrary Write

Buffer; stores input from stdin

‘strncpy’ copies the data from the user to the buffer the pointer points to

An innocuous buffer

Strategy is largely the same as previously shown regarding the pointer.

To determine if we can find a way to write to the buffer, attempt to constrain the buffer to equal our desired password.

Alternatively, constrain the buffer passed to `strncpy` as the `src` parameter.

The admin password (though now it’s ours!)
Simple Arbitrary Write Strategy

- Initialize state
- While we have not found a solution or exhaustively searched the binary:
  - For each active state:
    - If the program is calling `strncpy` (or `memcpy`, etc):
      - If the destination pointer and the source buffer are symbolic:
        - Constrain them to equal what we want
        - Solve for the user input
    - Step the active states
Simple Example: Arbitrary Jump

To determine if we can find a buffer overflow that would lead to an arbitrary jump, we could search for a situation where the return address of a function is unconstrained:

But there’s an easier way...
Simple Example: Arbitrary Jump

Search for a situation where the instruction pointer (ip) is symbolic:
Typically, when the instruction pointer becomes symbolic, Angr branches:

```python
# A simple guessing game.
user_input = raw_input('Enter the password: ')
if user_input == 'hunter2':
    print 'Success.'
else:
    print 'Try again.'
```

You could consider reaching both of these print statements as a single state, where the instruction pointer can take on multiple values: the address of “print ‘Success.’” or the address of “print ‘Try again.’”
Oh the Places You’ll Go!

However, when the instruction pointer is unconstrained, there are an infinite* number of possible branches. Execution cannot continue.

Normally, Angr throws away unconstrained states and continues with other paths that have a logical continuation.

In our case, we want to save them and determine if we can constrain the instruction pointer to equal the address of our malicious code.
Simple Arbitrary Jump Strategy

• Initialize state; instruct Angr to save unconstrained states
• While we have not found a solution or exhaustively searched the binary:
  • For each unconstrained state (commonly will be none):
    • If we can constrain the instruction pointer to what we want:
      • Constrain the instruction pointer
      • Solve for input
    • Step every active state
Not-so Automatic Exploit Generation

A few questions may come to mind:

• How do we know what we want to read/write/jump to?
• How do we determine if the computer is making an arbitrary read/write for the general case (strncpy isn’t used)?

Human intuition!

Researchers have only come up with mediocre algorithms for the general case (so far.) [citation needed]

If you find a better solution, publish it.
Complex Exploits

Real-world programs may not be exploitable by these simple approaches.

You may need to perform some combination of them, or take a different approach entirely.

To automatically do so is a very hard problem.
In some cases, symbolic execution can be a very powerful tool to automate vulnerability & bug discovery.

It is not a miracle algorithm that can autonomously discover any bug.

Nonetheless, understanding symbolic execution helps us understand the underlying concepts involved in exploit discovery, and gives us a powerful tool to use and research.