Fuzzing

```
1 var haystack = "foo";
2 var re_text = "^foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);
```
1. `var haystack = "foo";`
2. `var re_text = "^foo";`
3. `haystack += "x";`
4. `re_text += "(x)";`
5. `var re = new RegExp(re_text);`
6. `re.test(haystack);`
7. `RegExp.input = Number();`
8. `print(RegExp.$1);`
The Problem

```javascript
1 var haystack = "foo";
2 var re_text = "~foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);
```
The Problem

---

```javascript
var haystack = "foo";
var re_text = "~foo";
haystack += "x";
re_text += "(x)";
var re = new RegExp(re_text);
re.test(haystack);
RegExp.input = Number();
print(RegExp.$1);
```
The Problem

```javascript
var haystack = "foo";
var re_text = "~foo";
haystack += "x";
re_text += "(x)";
var re = new RegExp(re_text);
re.test(haystack);
RegExp.input = Number();
print(RegExp.$1);
```
The Problem

```javascript
var haystack = "foo";
var re_text = "~foo";
haystack += "x";
re_text += "(x)";
var re = new RegExp(re_text);
re.test(haystack);
RegExp.input = Number();
print(RegExp.$1);
```
The Problem

```
1 var haystack = "foo";
2 var re_text = "~foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);
```

How can we avoid this overhead?
The Idea

---

```javascript
var haystack = "foo";
var re_text = "^foo";
haystack += "x";
re_text += "(x)";
var re = new RegExp(re_text);
re.test(haystack);
RegExp.input = Number();
print(RegExp.$1);
```

---

system data
The Idea

1 var haystack = "foo";
2 var re_text = "^foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);

Figure 2: Test case generated by LangFuzz, crashing the

system data

Functions

unit data

System Input
Unit-Level Testing

• We can fuzz functions directly by generating function calls

• All we need is a grammar that produces the necessary code / calls / expressions
Example: urlparse()
Example: urlparse()

api_grammar = {
    "$START":
        ['$CALL'],
    '$CALL':
        ['urlparse("$URL")'],
    '$URL':
        ['$SCHEME://$AUTHORITY$PATH$QUERY'],
    ...}

– A call grammar
Example: urlparse()

api_grammar = {
    "$START":
        ["$CALL"],
    "$CALL":
        ['urlparse("$URL")'],
    "$URL":
        ["$SCHEME://$AUTHORITY$PATH$QUERY"],
    ...
}

for i in range(1, 100):
    call = produce(grammar)

- A call grammar
- Producing calls
Example: urlparse()

```
api_grammar = {
    "$START":
        ["$CALL"],
    "$CALL":
        ['urlparse("$URL")'],
    "$URL":
        ["$SCHEME://$AUTHORITY$PATH$QUERY"],
    ...
}

for i in range(1, 100):
    call = produce(grammar)

    urlparse("http://fuzzingbook.com:5591/x55")
    urlparse("https://user:password@cispa.saarland")
    ...
```

– A call grammar

– Producing calls
Example: urlparse()

api_grammar = {
    "$START":
        [$CALL],
    "$CALL":
        ['urlparse("$URL")'],
    "$URL":
        ['$SCHEME://$AUTHORITY$PATH$QUERY'],
    ...
}

for i in range(1, 100):
    call = produce(grammar)

    urlparse("http://fuzzingbook.com:5591/x55")
    urlparse("https://user:password@cispa.saarland")
    ...

    result = eval(call)
Demo
Whence the data?

Where do we get the data for arguments from?

- Idea #1: We can specify arguments as part of the generating grammar
- Problem: lots of manual work!
Whence the data?

Where do we get the data for arguments from?

- Idea #1: We can specify arguments as part of the generating grammar
- Problem: lots of manual work!

– Also, grammars may not be the best tool to express preconditions
Whence the data?

• Idea #2: We can detect static input types and call the function with random values

• Popular test generation technique, typically using sequences of calls (Randoop, EvoSuite)
Whence the data?

- Idea #2: We can detect static input types and call the function with random values
- Popular test generation technique, typically using sequences of calls (Randoop, EvoSuite)

- Actually, this would be hard for dynamically typed languages such as Python
False Failures

• Function may fail because generated arguments violate implicit preconditions

• Example: The generated call $\sqrt{-1}$ fails. Should this be fixed? Or $\text{open}(\text{NULL})$?

• Problem does not occur at system level, since the system must recognize and reject invalid inputs
False Failures

• Function may fail because generated arguments violate implicit preconditions

• Example: The generated call \texttt{sqrt}(-1) fails. Should this be fixed? Or \texttt{open(NULL)}?

• Problem does not occur at system level, since the system must recognize and reject invalid inputs

– which is why we prefer system fuzzing
Whence the data?

- Idea #3: We can take a system run and record function calls with all arguments.
- These can then be efficiently replayed.
Whence the data?

• Idea #3: We can take a system run and record function calls with all arguments

• These can then be efficiently replayed

– Realistic input data reduces the risk of false failures
Carving
- FSE 2006 -

Carving Differential Unit Test Cases from System Test Cases

Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, Jonathan Dokulil
Dept. of Computer Science and Engineering, University of Nebraska
Lincoln, NE, USA
{elbaum,hchin,dwyer,jdokulil}@cse.unl.edu

ABSTRACT
Unit test cases are focused and efficient. System tests are effective at exercising complex usage patterns. Differential unit tests (DUT) are a hybrid of unit and system tests. They are generated by carving the system components while executing a system test case, that influence the behavior of the target unit, and then re-assembling those components so that the unit can be exercised as it was by the system test. We conjecture that DUTs retain some of the advantages of unit tests, can be automatically and inexpensively generated, and have the potential for revealing faults related to intricate system executions. In this paper we present a framework for automatically carving and replaying DUTs that accounts for a wide-variety of strategies, we implement an instance of the framework with several techniques to mitigate test cost and enhance flexibility, and we empirically assess the efficacy of carving and replaying DUTs.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Reliability, Experimentation, Verification

Keywords
```
var haystack = "foo";
var re_text = "^foo";
haystack += "x";
re_text += "(x)";
var re = new RegExp(re_text);
re.test(haystack);
RegExp.input = Number();
print(RegExp.$1);
```
Carving

1 var haystack = "foo";
2 var re_text = "~foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);

system data
Carving

```
1 var haystack = "foo";
2 var re_text = "^foo";
3 haystack += "x";
4 re_text += "(x)";
5 var re = new RegExp(re_text);
6 re.test(haystack);
7 RegExp.input = Number();
8 print(RegExp.$1);
```

Figure 2: Test case generated by LangFuzz, crashing the

- System Input
- replay unit data
- Functions

system data
Recording Calls
# This is where we store all calls and arguments
the_args = {}

# Tracking function: Record all calls and all args
def traceit(frame, event, arg):
    if event == "call":
        code = frame.f_code
        function_name = code.co_name

        # When called, all arguments are local variables
        variables = frame.f_locals.keys()
        args = [(var, frame.f_locals[var]) for var in variables]

        if function_name not in the_args:
            the_args[function_name] = []
        if args not in the_args[function_name]:
            the_args[function_name].append(args)

    # If we return None, this will only be called for functions
    # (more efficient)
    return None
Demo
Replaying Calls
Replaying Calls

# Return function_name(arg[0], arg[1], ...) as a string
def call_with_args(function_name, args):
    return function_name + "(" + \
    ", " + repr(value) for (var, value) in args] + ")"

def do_call(call_string):
    try:
        result = eval(call_string)
    except:
        result = sys.exc_info()[0]
    return result

# Re-run all calls seen, invoking functions directly
def run_calls():
    for function_name in the_args.keys():
        for args in the_args[function_name]:
            call_string = call_with_args(function_name, args)
            do_call(call_string)
Demo
Fuzzing Functions

Since we have a collection of realistic data:

• We can also alter and recombine this data

• We obtain fuzzing at the function level!
Fuzzing Functions

Since we have a collection of realistic data:

- We can also alter and recombine this data
- We obtain fuzzing at the function level!

– Again, fuzzing functions is much more efficient
Fuzzing Functions

• Idea: We infer a grammar from the recorded calls
• Each argument (= each variable) gets an independent rule with alternative values (as with grammar mining for inputs)
• The grammar can then be used as a producer
Fuzzing Functions

• Idea: We infer a grammar from the recorded calls.
• Each argument (= each variable) gets an independent rule with alternative values (as with grammar mining for inputs).
• The grammar can then be used as a producer.

… and combined with search-based and symbolic testing.
Example: power()
Example: power()

```python
# Callable wrapper
def power(x, y):
    return math.pow(x, y)

for n in range(0, 10):
    x = power(n, n)
```
Example: power()

```python
# Callable wrapper
def power(x, y):
    return math.pow(x, y)

for n in range(0, 10):
    x = power(n, n)
```
Example: power()

```python
# Callable wrapper
def power(x, y):
    return math.pow(x, y)

for n in range(0, 10):
    x = power(n, n)

power(y = 0, x = 0)
power(y = 1, x = 1)
power(y = 2, x = 2)
power(y = 3, x = 3)
power(y = 4, x = 4)
power(y = 5, x = 5)
power(y = 6, x = 6)
power(y = 7, x = 7)
power(y = 8, x = 8)
power(y = 9, x = 9)
```
Example: power()

```python
# Callable wrapper
def power(x, y):
    return math.pow(x, y)

for n in range(0, 10):
    x = power(n, n)
```

```
- power(y = 0, x = 0)
- power(y = 1, x = 1)
- power(y = 2, x = 2)
- power(y = 3, x = 3)
- power(y = 4, x = 4)
- power(y = 5, x = 5)
- power(y = 6, x = 6)
- power(y = 7, x = 7)
- power(y = 8, x = 8)
- power(y = 9, x = 9)
```
Example: power()

# Callable wrapper
def power(x, y):
    return math.pow(x, y)

for n in range(0, 10):
    x = power(n, n)
    power(y = 0, x = 0)
    power(y = 1, x = 1)
    power(y = 2, x = 2)
    power(y = 3, x = 3)
    power(y = 4, x = 4)
    power(y = 5, x = 5)
    power(y = 6, x = 6)
    power(y = 7, x = 7)
    power(y = 8, x = 8)
    power(y = 9, x = 9)

$START ::= $POWER
$POWER ::= power(y =$POWER_Y, x =$POWER_X)
$POWER_Y ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
$POWER_X ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Mining Call Grammars
def mine_grammar_from_calls():
    all_calls = "$CALL"
    grammar = {
        "$START": [all_calls],
    }

    function_nonterminals = []
    for function_name in the_args.keys():
        nonterminal_name = nonterminal(function_name)
        function_nonterminals.append(nonterminal_name)

        # Add a rule for the function
        expansion = function_name + "("
        first_arg = True
        for (var, _) in the_args[function_name][0]:
            arg_name = nonterminal(function_name + "_" + var)
            if not first_arg:
                expansion += ", "
                first_arg = False
        expansion += var + " = " + arg_name
        expansion += ")"
        grammar[nonterminal_name] = [expansion]
# Add a rule for the function
expansion = function_name + "("
first_arg = True
for (var, _) in the_args[function_name][0]:
    arg_name = nonterminal(function_name + "_" + var)
    if not first_arg:
        expansion += ", "
        first_arg = False
    expansion += var + " = " + arg_name
expansion += ")"
grammar[nonterminal_name] = [expansion]

# Add rules for the arguments
values = {}
for args in the_args[function_name]:
    for (var, value) in args:
        if var not in values:
            values[var] = []
        if value not in values[var]:
            values[var].append(value)
g = value_rules(values, function_name)
for key in g.keys():
    grammar[key] = g[key]

# Add a rule for all calls
grammar[all_calls] = function_nonterminals

return grammar
Representing Values

- We assume that `repr(x)` can also be parsed back
Representing Values

- We assume that `repr(x)` can also be parsed back

```python
# Return a grammar only for the values in VALUES
def value_rules(values, prefix):
    grammar = {}
    for var in values.keys():
        arg_name = nonterminal(prefix + "_" + var)
        expansions = [repr(value) for value in values[var]]
        grammar[arg_name] = expansions
    return grammar
```
Demo
Fuzzing with Mined Call Grammars

- The mined grammar can be used for production just as a specified one
The mined grammar can be used for production just as a specified one

```python
for i in range(0, 100):
    call = produce(g)
    do_call(call)
```
Demo
More Ideas

• Not only recombine args, but also tuple elements, object attributes, etc. (see code for sample implementation)

• Combine with search-based or symbolic testing at the unit level
Even More Ideas

- Higher randomness in args increases the risk of false failures
- Integrate with system-level fuzzing:
  - provides realistic inputs to learn from
  - lifting unit-level args back to system allows for validation of features
Fuzzing + Carving

carve from generated system tests

generated
system data

Fuzzer

System Input

record unit data

Functions
Lifting
bring locally generated values back to system level

Fuzzer

generated system data

generated unit data

Functions

System Input
Lifting

bring locally generated values back to system level

Fuzzer

generated system data

System Input

generated unit data

Functions
Lifting

bring locally generated values back to system level
Lifting

bring locally generated values back to system level

generated unit data

Fuzzer

generated system data

System Input

Functions
Lifting
bring locally generated values back to system level

Fuzzer

generated system data

-1

System Input

generated unit data

Functions
Lifting

bring locally generated values back to system level

Fuzzer

System Input

Functions

generated system data
Lifting

bring locally generated values back to system level

Fuzzer

generated system data

System Input

-1

Functions
Lifting

bring locally generated values back to system level

Fuzzer

generated system data

System Input

−1

Functions
Lifting

bring locally generated values back to system level

Fuzzer

generated system data

System Input

Functions
Lifting

bring locally generated values back to system level

If the input with the lifted value reproduces the failure, we have a true failure
Alexander Kampmann
Hybrid testing
Automated Test Generation

Symbolic Execution

Program under Test

Fuzzing

Program under Test
Automated Test Generation

Symbolic Execution

Input Validation

Program under Test

Fuzzing

Input Validation

Program under Test
Automated Test Generation

Symbolic Execution

Fuzzing
Automated Test Generation

Symbolic Execution

Fuzzing

Input Validation

Program under Test
My Approach

Hybrid approach

use Fuzzing here

Input Validation

Program under Test
My Approach

Hybrid approach

use Fuzzing here

use Symbolic Execution here
A Sample Program

```c
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if(item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
```
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if(item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
A Sample Program

```c
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if(item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
```
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if(item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if(item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if (item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
System Overview

Fuzzing → Carving → Symbolic Analysis

Carving → Validation

Validation → Lifting
System Overview

Carving
Leaving the playground python world
Low-Level Virtual Machine

Abstraction

C | Rust | Haskell

clang | rustc | ghc

Low-Level Virtual Machine

llc

Machine Code
Carving in LLVM

array = [“one”, “two”, “three”]
Carving in LLVM

array = ["one", "two", "three"]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0C</td>
</tr>
<tr>
<td>0x02</td>
<td>0x10</td>
</tr>
<tr>
<td>0x03</td>
<td>0x06</td>
</tr>
<tr>
<td>0x04</td>
<td>0x14</td>
</tr>
<tr>
<td>0x05</td>
<td></td>
</tr>
<tr>
<td>0x06</td>
<td>'t'</td>
</tr>
<tr>
<td>0x07</td>
<td>'h'</td>
</tr>
<tr>
<td>0x08</td>
<td>'r'</td>
</tr>
<tr>
<td>0x09</td>
<td>'e'</td>
</tr>
<tr>
<td>0x0A</td>
<td>e'</td>
</tr>
<tr>
<td>0x0B</td>
<td>\0'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0C</td>
<td>'o'</td>
</tr>
<tr>
<td>0x0D</td>
<td>'n'</td>
</tr>
<tr>
<td>0x0E</td>
<td>'e'</td>
</tr>
<tr>
<td>0x0F</td>
<td>\0'</td>
</tr>
<tr>
<td>0x10</td>
<td>'t'</td>
</tr>
<tr>
<td>0x11</td>
<td>'w'</td>
</tr>
<tr>
<td>0x12</td>
<td>'o'</td>
</tr>
<tr>
<td>0x13</td>
<td>\0'</td>
</tr>
<tr>
<td>0x14</td>
<td>'h'</td>
</tr>
<tr>
<td>0x15</td>
<td>\0'</td>
</tr>
</tbody>
</table>
Carving in LLVM

array = ["one", "two", "three"]

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0x0C</td>
</tr>
<tr>
<td>0x02</td>
<td>0x10</td>
</tr>
<tr>
<td>0x03</td>
<td>0x06</td>
</tr>
<tr>
<td>0x04</td>
<td>0x14</td>
</tr>
<tr>
<td>0x05</td>
<td></td>
</tr>
<tr>
<td>0x06</td>
<td>'t'</td>
</tr>
<tr>
<td>0x07</td>
<td>'h'</td>
</tr>
<tr>
<td>0x08</td>
<td>'r'</td>
</tr>
<tr>
<td>0x09</td>
<td>'e'</td>
</tr>
<tr>
<td>0x0A</td>
<td>'e'</td>
</tr>
<tr>
<td>0x0B</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0C</td>
<td>'o'</td>
</tr>
<tr>
<td>0x0D</td>
<td>'n'</td>
</tr>
<tr>
<td>0x0E</td>
<td>'e'</td>
</tr>
<tr>
<td>0x0F</td>
<td>'\0'</td>
</tr>
<tr>
<td>0x10</td>
<td>'t'</td>
</tr>
<tr>
<td>0x11</td>
<td>'w'</td>
</tr>
<tr>
<td>0x12</td>
<td>'o'</td>
</tr>
<tr>
<td>0x13</td>
<td>'\0'</td>
</tr>
<tr>
<td>0x14</td>
<td>'h'</td>
</tr>
<tr>
<td>0x15</td>
<td>'\0'</td>
</tr>
</tbody>
</table>
Strings are zero-terminated

cJSON's parser:

```c
char * parse_string(char *str, char **ep)

{ "key": "value" }
```
Strings are zero-terminated

cJSON’s parser:

```c
char * parse_string(char *str, char **ep){
  "key": "value"
})
```

char * parse_string(char *str, char **ep)
Strings are zero-terminated

cJSON’s parser:

```c
char * parse_string(char *str, char **ep)
{  "key": "value"  }
"\"value\" \0"
```

the function returns the unmodified input
Strings are zero-terminated

cJSON’s parser:

```
char * parse_string(char *str, char **ep)
{
    "key": "value"
}
```

the function returns the unmodified input and the part of the string which it did not consume
Carving in LLVM

array = [“one”, “two”, “three”]
Carving in LLVM

array = [“one”, “two”, “three”]
Carving in LLVM

array = ["one", "two", "three"]

How do I know that cell 0x4 does not belong to the array?
Tracking malloc

array = malloc(sizeof(char *) * 3);
array[0] = "one";
array[1] = "two";
array[3] = "three";
Tracking malloc

array = malloc(sizeof(char *) * 3);
array[0] = “one”;  
array[1] = “two”;  
array[3] = “three”;  
Keep track of the allocation sizes!
Memory Sizes

- data type known:
  - pointer: 8 byte
  - int32_t: 4 byte
  - ...

- malloc’d memory:
  - keep track of malloc sizes

- string: zero-terminated (most of the time)

- memory provided by the operating system:
  - environment variables
  - main parameters
  - global variables
  - compile-time analysis
Carving

```c
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if(item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
```

Test Program randomly generated test input

```json
"M7NR": 1337
```
Carving

```c
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if (item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}
```
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if (item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}

int snippet() {
    cJSON_strcasecmp(     ,
                        "M7NR");
    cJSON_strcasecmp("M7NR", "testitem");
}
int main(int argc, char **argv) {
    char *text = readEntireFile(argv[1]);
    cJSON *json = cJSON_Parse(text);
    cJSON *item = cJSON_GetObjectItem(json, "testitem");
    if(item != NULL) {
        char *test = NULL;
        test[0] = 'c';
    }
}

int snippet() {
    cJSON_strcasecmp("testitem");
}

The Gap

System-level input

Unit-level input

Function Under Test
The Gap

System-level input

Unit-level input

Function Under Test

tainting
The Gap

System-level input

Unit-level input

tainting

Function Under Test

string equality
Mapping

```c
int snippet() {
    cJSON_strcasecmp("M7NR", "testitem");
}
```
Mapping

```c
int snippet() {
    cJSON_strcasecmp("M7NR", "testitem");
}
```
the symbolic execution will try to find a different value here
Symbolic Analysis

```c
int snippet() {
    cJSON_strcasecmp("M7NR", "testitem");
}
```

the symbolic execution will try to find a different value here
unit-level value

"testitem"
unit-level value

"testitem"
Lifting

{""M7NR"": 1337}
Lifting

{"M7NR": 1337}

"testitem"
Lifting

{""testitem"": 1337}
Lifting

{"testitem": 1337}
Validation

Input Validation

Program under Test
Validation

System Test (from Fuzzing)

Input Validation

Program under Test
Validation

System Test (from Fuzzing)

Carved Code Snippet

Input Validation

Program under Test
Validation

System Test (from Fuzzing)

Carved Code Snippet

Symbolic Execution predicts this path

Program under Test
Validation

System Test (from Fuzzing)

Carved Code Snippet

Symbolic Execution predicts this path

Actual execution of the lifted test
What if the path diverges?

• Target: Coverage
  • may still cover new code

• Target: Bugs
  • prediction not fulfilled: no bugs discovered
Questions?