

---

# **Whitesymex**

**Umut Barış Öztunç**

**May 30, 2021**



**CONTENTS:**

<b>1</b>	<b>Whitesymex</b>	<b>1</b>
1.1	Installation . . . . .	1
1.2	Usage . . . . .	1
1.3	Documentation . . . . .	3
<b>2</b>	<b>API Documentation</b>	<b>5</b>
2.1	whitesymex package . . . . .	5
<b>3</b>	<b>Indices and tables</b>	<b>17</b>
	<b>Python Module Index</b>	<b>19</b>
	<b>Index</b>	<b>21</b>



## WHITESYMEX

Whitesymex is a symbolic execution engine for [Whitespace](#)). It uses dynamic symbolic analysis to find execution paths of a Whitespace program. It is inspired by [angr](#).

## 1.1 Installation

It is available on pypi. It requires python 3.7.0+ to run.

```
$ pip install whitesymex
```

## 1.2 Usage

### 1.2.1 Command-line Interface

```
$ whitesymex -h
# usage: whitesymex [-h] [--version] [--find FIND] [--avoid AVOID] [--strategy {bfs,dfs,
↪random}]
#                               [--loop-limit LIMIT]
#                               file
#
# Symbolic execution engine for Whitespace.
#
# positional arguments:
#   file                  program to execute
#
# optional arguments:
#   -h, --help            show this help message and exit
#   --version             show program's version number and exit
#   --find FIND           string to find
```

(continues on next page)

(continued from previous page)

```
# --avoid AVOID      string to avoid
# --strategy {bfs,dfs,random}
#                   path exploration strategy (default: bfs)
# --loop-limit LIMIT maximum number of iterations for symbolic loops
```

Simple example:

```
$ whitesymex password_checker.ws --find 'Correct!' --avoid 'Nope.'
# p4ssw0rd
```

## 1.2.2 Python API

Simple example:

```
from whitesymex import parser
from whitesymex.path_group import PathGroup
from whitesymex.state import State

instructions = parser.parse_file("password_checker.ws")
state = State.create_entry_state(instructions)
path_group = PathGroup(state)
path_group.explore(find=b"Correct!", avoid=b"Nope.")
password = path_group.found[0].concretize()
print(password.decode())
# p4ssw0rd
```

More complex example from XCTF Finals 2020:

```
import z3

from whitesymex import parser, strategies
from whitesymex.path_group import PathGroup
from whitesymex.state import State

instructions = parser.parse_file("xctf-finals-2020-spaceship.ws")
flag_length = 18
flag = [z3.BitVec(f"flag_{i}", 24) for i in range(flag_length)] + list(b"\n")
state = State.create_entry_state(instructions, stdin=flag)

# The flag is printable.
for i in range(flag_length):
    state.solver.add(z3.And(0x20 <= flag[i], flag[i] <= 0x7F))

path_group = PathGroup(state)
path_group.explore(avoid=b"Imposter!", strategy=strategies.DFS)
flag = path_group.deadended[0].concretize()
print(flag.decode())
# xctf{Wh1t3sym3x!??}
```

You can also concretize your symbolic variables instead of stdin:

```
import z3

from whitesymex import parser
from whitesymex.path_group import PathGroup
from whitesymex.state import State

instructions = parser.parse_file("tests/data/xctf-finals-2020-spaceship.ws")
symflag = [z3.BitVec(f"flag_{i}", 24) for i in range(12)]
stdin = list(b"xctf{") + symflag + list(b"}\n")
state = State.create_entry_state(instructions, stdin=stdin)
for c in symflag:
    state.solver.add(z3.And(0x20 <= c, c <= 0x7F))
path_group = PathGroup(state)
path_group.explore(find=b"crewmember", avoid=b"Imposter!")
flag = path_group.found[0].concretize(symflag)
print("xctf{%s}" % flag)
# xctf{Wh1t3sym3x!??}
```

## 1.3 Documentation

The documentation is available at [whitesymex.readthedocs.io](https://whitesymex.readthedocs.io).





## API DOCUMENTATION

Information on specific functions, classes, and methods.

### 2.1 whitesymex package

#### 2.1.1 Subpackages

##### whitesymex.strategies package

**class** whitesymex.strategies.**BFS**(*path\_group: PathGroup, find: Callable[[State], bool], avoid: Callable[[State], bool], loop\_limit: Optional[int], num\_find: int = 1*)  
Bases: [whitesymex.strategies.strategy.Strategy](#)

**select\_states()**  
Selects states to be executed in the next iteration.  
This function is supposed to be implemented in subclasses.  
**Returns** A list of states to be executed.

**class** whitesymex.strategies.**DFS**(*path\_group: PathGroup, find: Callable[[State], bool], avoid: Callable[[State], bool], loop\_limit: Optional[int], num\_find: int = 1*)  
Bases: [whitesymex.strategies.strategy.Strategy](#)

**select\_states()**  
Selects states to be executed in the next iteration.  
This function is supposed to be implemented in subclasses.  
**Returns** A list of states to be executed.

**class** whitesymex.strategies.**Random**(*path\_group: PathGroup, find: Callable[[State], bool], avoid: Callable[[State], bool], loop\_limit: Optional[int], num\_find: int = 1*)  
Bases: [whitesymex.strategies.strategy.Strategy](#)

**select\_states()**  
Selects states to be executed in the next iteration.  
This function is supposed to be implemented in subclasses.  
**Returns** A list of states to be executed.

**class** whitesymex.strategies.**Strategy**(*path\_group: PathGroup, find: Callable[[State], bool], avoid: Callable[[State], bool], loop\_limit: Optional[int], num\_find: int = 1*)  
Bases: [object](#)

Strategy to select and step states during symbolic execution.

This class is supposed to be subclassed for each strategy.

**path\_group**

A PathGroup instance to run the strategy.

**find**

A filter function to decide if a state should be marked as found.

**avoid**

A filter function to decide if a state should be avoided.

**loop\_limit**

A maximum number of iterations for loops with a symbolic expression as a condition.

**loop\_counts**

A dict mapping ip values of conditionals to hit counts.

**num\_find**

Number of states to be found.

**run()**

Runs the symbolic execution with the strategy.

Returns if num\_find states are found. Otherwise, runs until no active states left.

**select\_states()** → list[State]

Selects states to be executed in the next iteration.

This function is supposed to be implemented in subclasses.

**Returns** A list of states to be executed.

**step(state: State)** → Optional[list[State]]

Steps the given state.

**The state is stepped until one of the followings happen:**

- The state exits.
- The state throws an error.
- The state gets marked as found.
- The state gets marked as avoided.
- The state returns multiple successor states.
- The state hits the loop limit.

**Parameters state** – A state to be stepped.

**Returns** A list of successor states is returned. If the state is classified such as errored, found, avoided, or hits the loop limit, None is returned.

## Submodules

### whitesymex.strategies.bfs module

```
class whitesymex.strategies.bfs.BFS(path_group: PathGroup, find: Callable[[State], bool], avoid:
                                     Callable[[State], bool], loop_limit: Optional[int], num_find: int = 1)
    Bases: whitesymex.strategies.strategy.Strategy
    select_states()
        Selects states to be executed in the next iteration.
        This function is supposed to be implemented in subclasses.
        Returns A list of states to be executed.
```

### whitesymex.strategies.dfs module

```
class whitesymex.strategies.dfs.DFS(path_group: PathGroup, find: Callable[[State], bool], avoid:
                                     Callable[[State], bool], loop_limit: Optional[int], num_find: int = 1)
    Bases: whitesymex.strategies.strategy.Strategy
    select_states()
        Selects states to be executed in the next iteration.
        This function is supposed to be implemented in subclasses.
        Returns A list of states to be executed.
```

### whitesymex.strategies.random module

```
class whitesymex.strategies.random.Random(path_group: PathGroup, find: Callable[[State], bool], avoid:
                                           Callable[[State], bool], loop_limit: Optional[int], num_find:
                                           int = 1)
    Bases: whitesymex.strategies.strategy.Strategy
    select_states()
        Selects states to be executed in the next iteration.
        This function is supposed to be implemented in subclasses.
        Returns A list of states to be executed.
```

### whitesymex.strategies.strategy module

```
class whitesymex.strategies.strategy.Strategy(path_group: PathGroup, find: Callable[[State], bool],
                                              avoid: Callable[[State], bool], loop_limit:
                                              Optional[int], num_find: int = 1)
    Bases: object
    Strategy to select and step states during symbolic execution.
    This class is supposed to be subclassed for each strategy.
    path_group
        A PathGroup instance to run the strategy.
```

**find**

A filter function to decide if a state should be marked as found.

**avoid**

A filter function to decide if a state should be avoided.

**loop\_limit**

A maximum number of iterations for loops with a symbolic expression as a condition.

**loop\_counts**

A dict mapping ip values of conditionals to hit counts.

**num\_find**

Number of states to be found.

**run()**

Runs the symbolic execution with the strategy.

Returns if num\_find states are found. Otherwise, runs until no active states left.

**select\_states()** → list[State]

Selects states to be executed in the next iteration.

This function is supposed to be implemented in subclasses.

**Returns** A list of states to be executed.

**step**(state: State) → Optional[list[State]]

Steps the given state.

**The state is stepped until one of the followings happen:**

- The state exits.
- The state throws an error.
- The state gets marked as found.
- The state gets marked as avoided.
- The state returns multiple successor states.
- The state hits the loop limit.

**Parameters** **state** – A state to be stepped.

**Returns** A list of successor states is returned. If the state is classified such as errored, found, avoided, or hits the loop limit, None is returned.

**whitesymex.strategies.strategy.is\_symbolic\_conditional**(state: State) → bool

Checks whether a state is on a symbolic conditional instruction.

A symbolic conditional is a conditional instruction that contains symbolic expressions in its condition.

**Parameters** **state** – A state to be checked.

**Returns** True if the current instruction is a symbolic conditional. Otherwise, returns False.

## 2.1.2 Submodules

### whitesymex.cli module

`whitesymex.cli.main()`

`whitesymex.cli.parse_args()` → `argparse.Namespace`

`whitesymex.cli.str_to_strategy(s: str)` → `Type[whitesymex.strategies.strategy.Strategy]`

### whitesymex.errors module

**exception** `whitesymex.errors.DivideByZeroError`

Bases: `whitesymex.errors.SymbolicExecutionError`, `ZeroDivisionError`

**exception** `whitesymex.errors.EmptyCallstackError`

Bases: `whitesymex.errors.SymbolicExecutionError`

**exception** `whitesymex.errors.EmptyStackError`

Bases: `whitesymex.errors.SymbolicExecutionError`

**exception** `whitesymex.errors.ParameterDecodeError`

Bases: `whitesymex.errors.ParserError`

**exception** `whitesymex.errors.ParserError`

Bases: `whitesymex.errors.WhitesymexError`

**exception** `whitesymex.errors.SolverError`

Bases: `whitesymex.errors.WhitesymexError`

**exception** `whitesymex.errors.StrategyError`

Bases: `whitesymex.errors.WhitesymexError`

**exception** `whitesymex.errors.StrategyNotImplementedError`

Bases: `whitesymex.errors.StrategyError`, `NotImplementedError`

**exception** `whitesymex.errors.SymbolicExecutionError`

Bases: `whitesymex.errors.WhitesymexError`

**exception** `whitesymex.errors.UnknownIMPErrors`

Bases: `whitesymex.errors.ParserError`

**exception** `whitesymex.errors.UnknownOpError`

Bases: `whitesymex.errors.ParserError`

**exception** `whitesymex.errors.UnknownParameterError`

Bases: `whitesymex.errors.ParserError`

**exception** `whitesymex.errors.WhitesymexError`

Bases: `Exception`

### whitesymex.imp module

```
class whitesymex.imp.IMP(value)
    Bases: enum.Enum

    Instruction Modification Parameters (IMP) for Whitespace.

    op_type
        Respective ops.Op subclass for the IMP value.

    pattern
        A string pattern to match the IMP.

    ARITHMETIC = '\t '
    FLOW_CONTROL = '\n'
    HEAP_ACCESS = '\t\t'
    IO = '\t\n'
    STACK_MANIPULATION = ' '
```

### whitesymex.instruction module

```
class whitesymex.instruction.Instruction(imp: 'IMP', op: 'Op', parameter: 'Optional[int]')
    Bases: object

    imp: IMP
    op: Op
    parameter: Optional[int]
```

### whitesymex.ops module

```
class whitesymex.ops.ArithmeticOp(value)
    Bases: whitesymex.ops.Op
    An enumeration.

    ADD = ' '
    DIV = '\t '
    MOD = '\t\t'
    MUL = ' \n'
    SUB = ' \t'

class whitesymex.ops.FlowControlOp(value)
    Bases: whitesymex.ops.Op
    An enumeration.

    CALL = ' \t'
    EXIT = '\n\n'
    JUMP = ' \n'
    JUMP_IF_NEGATIVE = '\t\t'
```

```
JUMP_IF_ZERO = '\t '
MARK = ' '
RETURN = '\t\n'

class whitesymex.ops.HeapAccessOp(value)
    Bases: whitesymex.ops.Op
    An enumeration.
    RETRIEVE = '\t'
    STORE = ' '

class whitesymex.ops.IOOp(value)
    Bases: whitesymex.ops.Op
    An enumeration.
    PRINT_CHAR = ' '
    PRINT_NUMBER = ' \t'
    READ_CHAR = '\t '
    READ_NUMBER = '\t\t'

class whitesymex.ops.Op(value)
    Bases: enum.Enum
    Op values for Whitespace commands.

    parameter
        A Parameter value that represents the parameter type for the op. If the op does not take any parameters,
        this value is None.

    pattern
        A string pattern to match the op.

class whitesymex.ops.StackManipulationOp(value)
    Bases: whitesymex.ops.Op
    An enumeration.
    COPY_TO_TOP = '\t '
    DISCARD_TOP = '\n\n'
    DUP_TOP = '\n '
    PUSH = ' '
    SLIDE_N_OFF = '\t\n'
    SWAP_TOP2 = '\n\t'
```

### whitesymex.parameter module

**class** whitesymex.parameter.**Parameter**(*value*)

Bases: enum.Enum

Parameter types for the commands.

**pattern**

A string regex pattern to match the parameter.

**LABEL** = '([\t ]+)\n'

**NUMBER** = '([\t ]+)\n'

### whitesymex.parser module

whitesymex.parser.**parse\_code**(*code: str*) → list[Instruction]

Parses the given code string to list of instructions.

whitesymex.parser.**parse\_file**(*filename: str*) → list[Instruction]

Reads and parses the given file to list of instructions.

### whitesymex.path\_group module

**class** whitesymex.path\_group.**PathGroup**(*state: State*)

Bases: object

Organizes states into stashes for symbolic execution.

**active**

A list of states that are still active.

**deadended**

A list of states that are exited gracefully.

**avoided**

A list of avoided states.

**found**

A list of found states.

**errored**

A list of states that encounter an error during execution.

**explore**(*find: Optional[Union[bytes, Callable[[State], bool]]] = None, avoid: Optional[Union[bytes, Callable[[State], bool]]] = None, strategy: type[strategies.Strategy] = <class 'whitesymex.strategies.bfs.BFS'>, loop\_limit: Optional[int] = None, num\_find: int = 1)*

Explores the active states and updates stashes accordingly.

It returns when there is no active states left or num\_find states are found.

#### Parameters

- **find** – Either bytes that are expected to be found in a state's stdout or a function that accepts a state and returns True if the state shall be classified as found.
- **avoid** – Either bytes that are expected to be avoided in a state's stdout or a function that accepts a state and returns True if the state shall be classified as avoided.
- **strategy** – A strategies.Strategy subclass that will be used to select states at each iteration.



- **loop\_limit** – A maximum limit for loops with a symbolic expression as its condition.
- **num\_find** – Number of states to be found.

**Raises** *StrategyError* – The given strategy is not a subclass of `strategies.Strategy` class.

`whitesymex.path_group.condition_to_lambda(condition: Optional[Union[bytes, Callable[[State], bool]]],  
default: bool = False) → Callable[[State], bool]`

Converts condition to lambda function that returns True or False.

#### Parameters

- **condition** – A condition can be a function or bytes that are expected to be found in a state's stdout.
- **default** – A bool value to be returned by lambda function if the condition is None.

**Returns** A lambda function that accepts a state as parameter that returns True if the condition is satisfied.

## whitesymex.solver module

`class whitesymex.solver.Solver`

Bases: `object`

SMT solver to solve path constraints.

#### constraints

A list of path constraints.

#### store

A dict mapping symbolic variables to concrete values.

`add(constraints: Union[z3.BoolRef, list[z3.BoolRef]])`

Adds constraints to the solver.

**Parameters** **constraints** – A single constraint or a list of constraints to add.

#### clone()

Returns a copy of the solver.

Both constraints and store are shallow-copied.

`eval(expression: z3.z3.ExprRef) → Union[bool, int]`

Evaluates an expression using the concrete values from the store.

**Parameters** **expression** – An expression to be evaluated.

**Returns** The evaluated value of the expression.

**Raises** *SolverError* – Failed to evaluate the expression as int or bool.

`is_satisfiable() → bool`

Checks and returns if the constraints are satisfiable.

`simplify(expression: z3.z3.ExprRef) → z3.z3.ExprRef`

Simplifies an expression by substituting concrete values.

Only the variables that exist in store are substituted.

**Parameters** **expression** – An expression to be simplified.

**Returns** The substituted and simplified expression.

**whitesymex.state module**

**class** whitesymex.state.**State**(*instructions: list[Instruction], labels: dict[int, int], stdin: Optional[deque[Value]], bitlength: int*)

Bases: object

Represents the execution state.

**ip**

An integer representing instruction pointer/program counter.

**stack**

A deque for the execution stack.

**callstack**

A deque that stores return addresses as stack.

**heap**

A dictionary to store/retrieve values by indexes.

**labels**

A dictionary that maps labels to ip values.

**instructions**

A list that contains program instructions.

**input**

A deque to represent stdin. As long as this deque is not empty, inputs will be read from it. However, if it is empty, symbolic variables will be read automatically.

**stdin**

A list that contains inputs read so far.

**stdout**

A list that represents stdout.

**var\_to\_type**

A dictionary that maps variables to VarType values.

**solver**

A whitesymex.solver.Solver instance to store and solve path constraints.

**operations**

A dictionary that maps whitesymex.ops.Op values to respective methods.

**clone()** → *whitesymex.state.State*

Returns a copy of the state.

Properties are shallow copied except for instructions and labels which are not copied but shared between the clones.

**concretize**(*buffer: list[Union[int, z3.ExprRef]] = None*) → bytes

Converts given symbolic buffer to concrete bytes.

If the buffer is None, it concretizes stdin instead.

**Parameters** **buffer** – A list that contains either bytes or symbolic variables.

**Returns** Concretized buffer as bytes object.

**classmethod** **create\_entry\_state**(*instructions: list[Instruction], stdin: list[Value] = None, bitlength: int = 24*) → *State*

Returns an entry state for the Whitespace program.

**Parameters**

- **instructions** – A list of instructions.
- **stdin** – A list of ints or symbolic variables.
- **bitlength** – Length of symbolic bitvectors. If this value is None, unbounded symbolic integers are used instead of bitvectors.

**property instruction:** `Optional[Instruction]`

Current instruction pointed by ip.

If the ip points to a location that is out of program's space, None is returned.

**is\_satisfiable()** → bool

Returns whether the path constraints are satisfiable or not.

**step()** → list[*State*]

Single-steps the current state.

If the instruction is conditional and the condition is a symbolic expression, the state clones itself and single-steps both paths.

**Returns** A list that contains the successor states.

**class** whitesymex.state.**VarType**(*value*)

Bases: `enum.Enum`

An enumeration.

**CHAR** = 1

**NUMBER** = 2



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### W

- `whitesymex`, 5
- `whitesymex.cli`, 9
- `whitesymex.errors`, 9
- `whitesymex.imp`, 10
- `whitesymex.instruction`, 10
- `whitesymex.ops`, 10
- `whitesymex.parameter`, 12
- `whitesymex.parser`, 12
- `whitesymex.path_group`, 12
- `whitesymex.solver`, 13
- `whitesymex.state`, 14
- `whitesymex.strategies`, 5
- `whitesymex.strategies.bfs`, 7
- `whitesymex.strategies.dfs`, 7
- `whitesymex.strategies.random`, 7
- `whitesymex.strategies.strategy`, 7





## A

active (*whitesymex.path\_group.PathGroup* attribute), 12  
 ADD (*whitesymex.ops.ArithmeticOp* attribute), 10  
 add() (*whitesymex.solver.Solver* method), 13  
 ARITHMETIC (*whitesymex.imp.IMP* attribute), 10  
 ArithmeticOp (class in *whitesymex.ops*), 10  
 avoid (*whitesymex.strategies.Strategy* attribute), 6  
 avoid (*whitesymex.strategies.strategy.Strategy* attribute), 8  
 avoided (*whitesymex.path\_group.PathGroup* attribute), 12

## B

BFS (class in *whitesymex.strategies*), 5  
 BFS (class in *whitesymex.strategies.bfs*), 7

## C

CALL (*whitesymex.ops.FlowControlOp* attribute), 10  
 callstack (*whitesymex.state.State* attribute), 14  
 CHAR (*whitesymex.state.VarType* attribute), 15  
 clone() (*whitesymex.solver.Solver* method), 13  
 clone() (*whitesymex.state.State* method), 14  
 concretize() (*whitesymex.state.State* method), 14  
 condition\_to\_lambda() (in module *whitesymex.path\_group*), 13  
 constraints (*whitesymex.solver.Solver* attribute), 13  
 COPY\_TO\_TOP (*whitesymex.ops.StackManipulationOp* attribute), 11  
 create\_entry\_state() (*whitesymex.state.State* class method), 14

## D

deadended (*whitesymex.path\_group.PathGroup* attribute), 12  
 DFS (class in *whitesymex.strategies*), 5  
 DFS (class in *whitesymex.strategies.dfs*), 7  
 DISCARD\_TOP (*whitesymex.ops.StackManipulationOp* attribute), 11  
 DIV (*whitesymex.ops.ArithmeticOp* attribute), 10  
 DivideByZeroError, 9

DUP\_TOP (*whitesymex.ops.StackManipulationOp* attribute), 11

## E

EmptyCallstackError, 9  
 EmptyStackError, 9  
 errored (*whitesymex.path\_group.PathGroup* attribute), 12  
 eval() (*whitesymex.solver.Solver* method), 13  
 EXIT (*whitesymex.ops.FlowControlOp* attribute), 10  
 explore() (*whitesymex.path\_group.PathGroup* method), 12

## F

find (*whitesymex.strategies.Strategy* attribute), 6  
 find (*whitesymex.strategies.strategy.Strategy* attribute), 7  
 FLOW\_CONTROL (*whitesymex.imp.IMP* attribute), 10  
 FlowControlOp (class in *whitesymex.ops*), 10  
 found (*whitesymex.path\_group.PathGroup* attribute), 12

## H

heap (*whitesymex.state.State* attribute), 14  
 HEAP\_ACCESS (*whitesymex.imp.IMP* attribute), 10  
 HeapAccessOp (class in *whitesymex.ops*), 11

## I

IMP (class in *whitesymex.imp*), 10  
 imp (*whitesymex.instruction.Instruction* attribute), 10  
 input (*whitesymex.state.State* attribute), 14  
 Instruction (class in *whitesymex.instruction*), 10  
 instruction (*whitesymex.state.State* property), 15  
 instructions (*whitesymex.state.State* attribute), 14  
 IO (*whitesymex.imp.IMP* attribute), 10  
 IOOp (class in *whitesymex.ops*), 11  
 ip (*whitesymex.state.State* attribute), 14  
 is\_satisfiable() (*whitesymex.solver.Solver* method), 13  
 is\_satisfiable() (*whitesymex.state.State* method), 15  
 is\_symbolic\_conditional() (in module *whitesymex.strategies.strategy*), 8

**J**

JUMP (*whitesymex.ops.FlowControlOp* attribute), 10  
JUMP\_IF\_NEGATIVE (*whitesymex.ops.FlowControlOp* attribute), 10  
JUMP\_IF\_ZERO (*whitesymex.ops.FlowControlOp* attribute), 10

**L**

LABEL (*whitesymex.parameter.Parameter* attribute), 12  
labels (*whitesymex.state.State* attribute), 14  
loop\_counts (*whitesymex.strategies.Strategy* attribute), 6  
loop\_counts (*whitesymex.strategies.strategy.Strategy* attribute), 8  
loop\_limit (*whitesymex.strategies.Strategy* attribute), 6  
loop\_limit (*whitesymex.strategies.strategy.Strategy* attribute), 8

**M**

main() (*in module whitesymex.cli*), 9  
MARK (*whitesymex.ops.FlowControlOp* attribute), 11  
MOD (*whitesymex.ops.ArithmeticOp* attribute), 10  
module  
    whitesymex, 5  
    whitesymex.cli, 9  
    whitesymex.errors, 9  
    whitesymex.imp, 10  
    whitesymex.instruction, 10  
    whitesymex.ops, 10  
    whitesymex.parameter, 12  
    whitesymex.parser, 12  
    whitesymex.path\_group, 12  
    whitesymex.solver, 13  
    whitesymex.state, 14  
    whitesymex.strategies, 5  
    whitesymex.strategies.bfs, 7  
    whitesymex.strategies.dfs, 7  
    whitesymex.strategies.random, 7  
    whitesymex.strategies.strategy, 7  
MUL (*whitesymex.ops.ArithmeticOp* attribute), 10

**N**

num\_find (*whitesymex.strategies.Strategy* attribute), 6  
num\_find (*whitesymex.strategies.strategy.Strategy* attribute), 8  
NUMBER (*whitesymex.parameter.Parameter* attribute), 12  
NUMBER (*whitesymex.state.VarType* attribute), 15

**O**

Op (*class in whitesymex.ops*), 11  
op (*whitesymex.instruction.Instruction* attribute), 10  
op\_type (*whitesymex.imp.IMP* attribute), 10  
operations (*whitesymex.state.State* attribute), 14

**P**

Parameter (*class in whitesymex.parameter*), 12  
parameter (*whitesymex.instruction.Instruction* attribute), 10  
parameter (*whitesymex.ops.Op* attribute), 11  
ParameterDecodeError, 9  
parse\_args() (*in module whitesymex.cli*), 9  
parse\_code() (*in module whitesymex.parser*), 12  
parse\_file() (*in module whitesymex.parser*), 12  
ParserError, 9  
path\_group (*whitesymex.strategies.Strategy* attribute), 6  
path\_group (*whitesymex.strategies.strategy.Strategy* attribute), 7  
PathGroup (*class in whitesymex.path\_group*), 12  
pattern (*whitesymex.imp.IMP* attribute), 10  
pattern (*whitesymex.ops.Op* attribute), 11  
pattern (*whitesymex.parameter.Parameter* attribute), 12  
PRINT\_CHAR (*whitesymex.ops.IOOp* attribute), 11  
PRINT\_NUMBER (*whitesymex.ops.IOOp* attribute), 11  
PUSH (*whitesymex.ops.StackManipulationOp* attribute), 11

**R**

Random (*class in whitesymex.strategies*), 5  
Random (*class in whitesymex.strategies.random*), 7  
READ\_CHAR (*whitesymex.ops.IOOp* attribute), 11  
READ\_NUMBER (*whitesymex.ops.IOOp* attribute), 11  
RETRIEVE (*whitesymex.ops.HeapAccessOp* attribute), 11  
RETURN (*whitesymex.ops.FlowControlOp* attribute), 11  
run() (*whitesymex.strategies.Strategy* method), 6  
run() (*whitesymex.strategies.strategy.Strategy* method), 8

**S**

select\_states() (*whitesymex.strategies.BFS* method), 5  
select\_states() (*whitesymex.strategies.bfs.BFS* method), 7  
select\_states() (*whitesymex.strategies.DFS* method), 5  
select\_states() (*whitesymex.strategies.dfs.DFS* method), 7  
select\_states() (*whitesymex.strategies.Random* method), 5  
select\_states() (*whitesymex.strategies.random.Random* method), 7  
select\_states() (*whitesymex.strategies.Strategy* method), 6  
select\_states() (*whitesymex.strategies.strategy.Strategy* method), 8  
simplify() (*whitesymex.solver.Solver* method), 13  
SLIDE\_N\_OFF (*whitesymex.ops.StackManipulationOp* attribute), 11  
Solver (*class in whitesymex.solver*), 13

solver (*whitesymex.state.State* attribute), 14  
 SolverError, 9  
 stack (*whitesymex.state.State* attribute), 14  
 STACK\_MANIPULATION (*whitesymex.imp.IMP* attribute), 10  
 StackManipulationOp (*class in whitesymex.ops*), 11  
 State (*class in whitesymex.state*), 14  
 stdin (*whitesymex.state.State* attribute), 14  
 stdout (*whitesymex.state.State* attribute), 14  
 step() (*whitesymex.state.State* method), 15  
 step() (*whitesymex.strategies.Strategy* method), 6  
 step() (*whitesymex.strategies.strategy.Strategy* method), 8  
 STORE (*whitesymex.ops.HeapAccessOp* attribute), 11  
 store (*whitesymex.solver.Solver* attribute), 13  
 str\_to\_strategy() (*in module whitesymex.cli*), 9  
 Strategy (*class in whitesymex.strategies*), 5  
 Strategy (*class in whitesymex.strategies.strategy*), 7  
 StrategyError, 9  
 StrategyNotImplementedError, 9  
 SUB (*whitesymex.ops.ArithmeticOp* attribute), 10  
 SWAP\_TOP2 (*whitesymex.ops.StackManipulationOp* attribute), 11  
 SymbolicExecutionError, 9

## U

UnknownIMPErrors, 9  
 UnknownOpError, 9  
 UnknownParameterError, 9

## V

var\_to\_type (*whitesymex.state.State* attribute), 14  
 VarType (*class in whitesymex.state*), 15

## W

whitesymex  
   module, 5  
 whitesymex.cli  
   module, 9  
 whitesymex.errors  
   module, 9  
 whitesymex.imp  
   module, 10  
 whitesymex.instruction  
   module, 10  
 whitesymex.ops  
   module, 10  
 whitesymex.parameter  
   module, 12  
 whitesymex.parser  
   module, 12  
 whitesymex.path\_group  
   module, 12  
 whitesymex.solver

  module, 13  
 whitesymex.state  
   module, 14  
 whitesymex.strategies  
   module, 5  
 whitesymex.strategies.bfs  
   module, 7  
 whitesymex.strategies.dfs  
   module, 7  
 whitesymex.strategies.random  
   module, 7  
 whitesymex.strategies.strategy  
   module, 7  
 WhitesymexError, 9