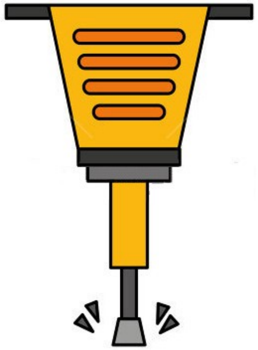
Reverse Engineering

Class 6

Fuzzing

Fuzzing

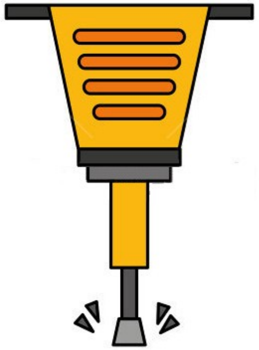


- Grey box testing
  - Source code access is not necessary. If available, useful but full understanding is not required
  - May be guided by reverse engineering
- Send, in an automatized way, valid and invalid inputs to an application with the goal of triggering bad behavior
  - Eventually, security problems
- Find vulnerabilities (bug hunting)
  - Internally
  - Externally (bug bounty, security advisory, research)

Fuzzing

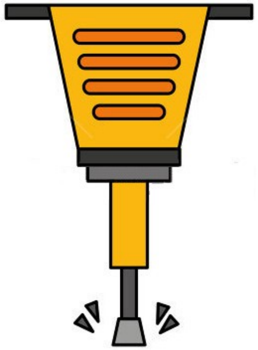
- Applicable to all types of inputs:
  - Web applications
    - POST/GET parameters fuzzing
  - File formats (doc, jpg, mp3, etc.) and file systems
    - Vulnerabilities in the parser
  - Network protocols
  - Programming languages
    - I.e. JavaScript can be seen as a complex input for a browser
  - Drivers
    - I.e. *ioctls* handled by a driver, file system/network filters, read/write operations in a char device, etc.
  - Etc.

Fuzzing



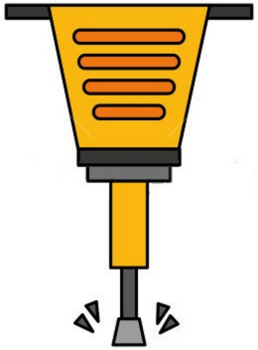
- Relevance of fuzzing
  - Relatively new discipline
  - Significant industry effort
    - ClusterFuzz, OSS-Fuzz (Google)
    - SAGE (Microsoft)
  - Yet much to be done
  - Relevant because of the number of vulnerabilities found

Fuzzing



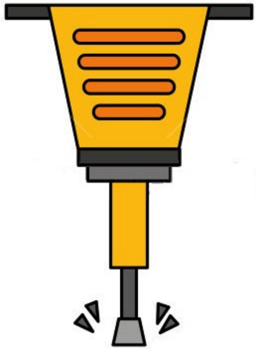
- Relevance of fuzzing
  - Commercial and open fuzzers
    - PeachFuzzer (commercial)
    - SPIKE (open)
    - AFL (open)
  - Generic fuzzing frameworks
  - Custom fuzzers (ad-hoc)

Fuzzing



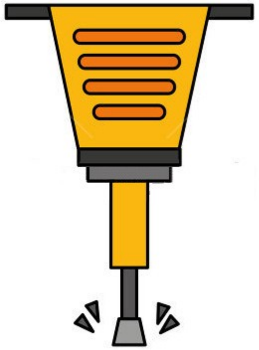
- Limits of fuzzing
  - Logic bugs or data attacks
    - Fuzzers are generally not focused on logic bugs like information disclosure or privilege escalation
  - Memory corruption bugs that do not cause crashes
    - It's necessary to recompile with libraries (or compilation flags) that set sentinels around buffers to expose memory corruptions
  - Race conditions
    - Difficult to reproduce bugs

Fuzzing

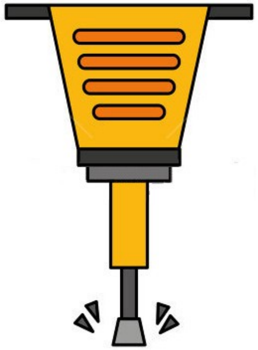


- Types of fuzzers
  - Purely random fuzzers
    - Generate garbage inputs
    - No cost but dumb
  - Mutational
    - Valid inputs are randomly modified (I.e. mutations, permutations, replacements with dictionaries or magic numbers)
    - It’s important to have a representative set of inputs

Fuzzing



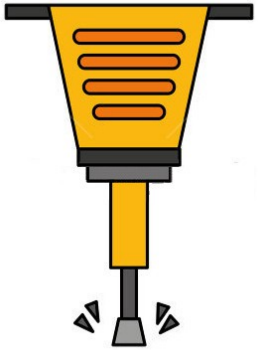
- Types of fuzzers
  - Evolutionary or genetic
    - Mutational variant, generation is guided by metrics and feedback
  - Generational
    - Inputs are generated based on a model or specification (I.e. language grammar or communications protocol)
    - High development cost. Specification is not always available. It may be necessary to do reverse engineering
  - Mixed



Fuzzing

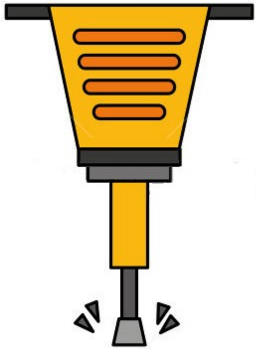
- Metrics
  - Exercise the highest number of possible execution flows and memory states
    - *Code-coverage*
  - Performance
  - Reliable crash detection
  - Reproducible cases (documented)

Fuzzing



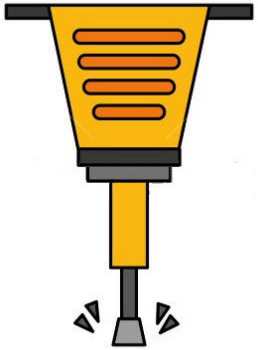
- Stages
  - Inputs identification and format analysis
    - Not always obvious:
      - Sockets?
      - Syscalls?
      - Files? Meta-data?
      - Environment variables? Which?
      - Registry? Which key?
      - IPC mechanisms?

Fuzzing



- Stages
  - Automated and fast input generation
  - Automation
    - Fast sending of inputs
    - Reliable crash detection
  - Crash analysis
    - Reduction of inputs that generate crashes (manual or automated)
    - Exploitability analysis (manual)

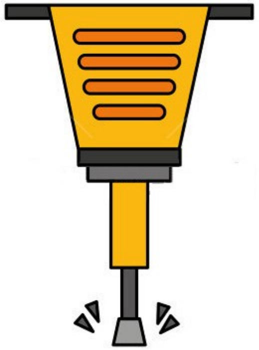
Fuzzing



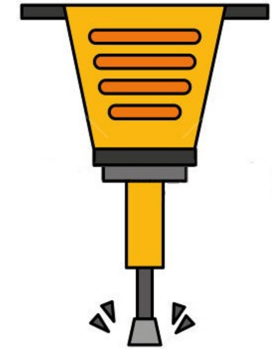
- Purely random fuzzers problem

Demo 6.1

Fuzzing



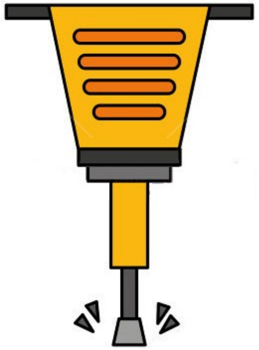
- Inputs format analysis
  - Key-value fields (I.e. JSON, HTTP header)
  - Variable length fields
  - Fields bounded by special characters
  - Text inputs (ASCII, UTF-8) or binary inputs
  - Understanding inputs format may help to better focus the effort. Sometimes, inputs analysis requires reverse engineering



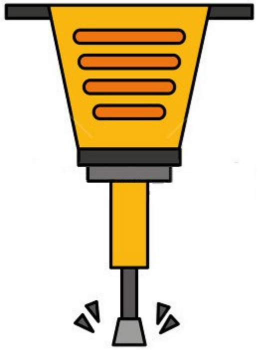
Fuzzing

- Assume that an application receives a 64 bit integer as input
  - Trying the whole range has a high computational and time cost
  - Is possible to build a smarter fuzzer? Which heuristics can be applied to this case?

Fuzzing



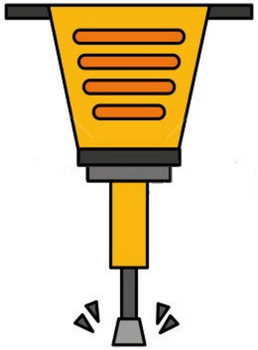
- Range boundaries, assuming different sizes to represent an integer:
  - 0...0xFF, 0...0xFF, 0...0xFFFF, 0...0xFFFFFFFF, 0…0xFFFFFFFFFFFFFFFF
  - What would happen if the integer is added to a constant? (I.e. for memory allocation)
  - Test values near boundaries:
    - 0, 1, 2, 3, 4 … 0xFD, 0xFE, 0xFF, etc.



Fuzzing

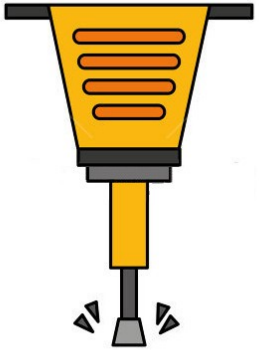
- What if the integer is multiplied by a constant? (I.e. 2)
  - Test range boundaries divided by the constant and near values. I.e.: 0xFF/2, 0xFFFF/2, 0xFFFFFFFF/2, etc.
- Test magic numbers
  - Integers that may have a special meaning within a context (I.e. constants, enumerative values)

Fuzzing



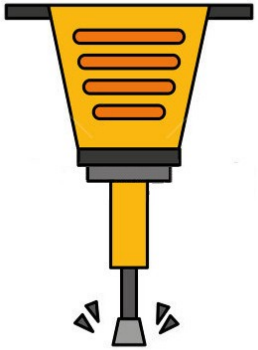
- Assume that an application receives a string as input
  - Which heuristics can be applied here?

Fuzzing



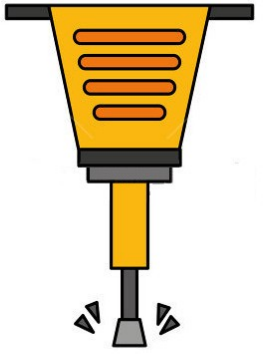
- Different encodings and multi-byte characters
  - ASCII, UTF-8, UTF-16, UTF-32, html encoding, etc.
  - Are there format conversions? Are implementations correct? Are there problems calculating lengths?
- Escape characters, delimiters, special characters according to the context. I.e. if an XML parser is being tested, it makes sense to try characters like “<” and sequences like “<![CDATA[]]>”.
- Null terminated strings? Has string data type a length at the beginning? (I.e. BSTR)
- Delimiter characters repetition (is it possible to trigger an overflow in a variable?)
- Different lengths
- Format strings (“%s, %d ...”)
- Dictionary words (according to the context)

Fuzzing



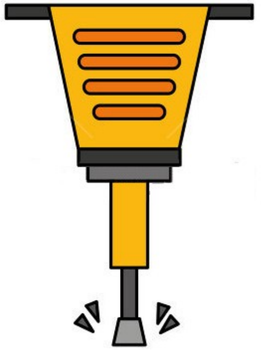
- In-memory fuzzing
  - Inputs are directly injected into the targeted process memory
    - How can it be done?

Fuzzing

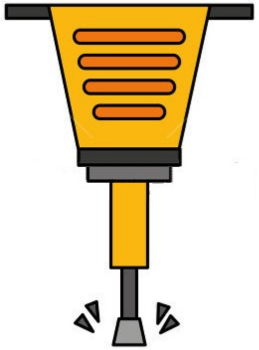


- In-memory fuzzing
  - Improve performance
  - Avoid generated input post-processing
    - Encrypt, sign, calculate checksums, include a previous token or other integrity control, etc.
  - Skip previous states in the state machine
    - I.e. authentication

Fuzzing



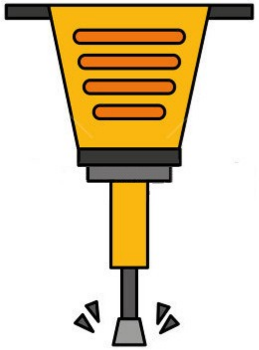
- In-memory fuzzing
  - Higher implementation cost
  - It’s necessary to start from a valid memory state (one that can be reached through a sequence of valid inputs)
    - This does not prevent from false positives. I.e. a previous filter or check may discard the input that generates the crash



Fuzzing

- In-memory fuzzing
  - Patch process memory to execute trampolines (hooks)
    - How?
  - Binary instrumentation frameworks
    - DynamoRIO
    - PIN
  - Recompile with hooks (if source code is available)

Fuzzing

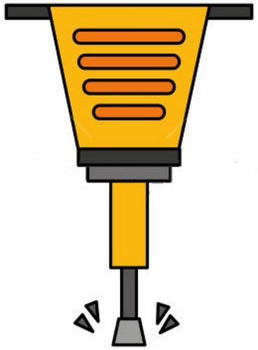


- Automation
  - Automation is everything
  - Computing cost is low compared to qualified talent
  - The number of cases that can be tested by unit of time is significantly higher, and cases can be tried on multiple targets
  - Focus efforts on a good case generation and execution

Fuzzing

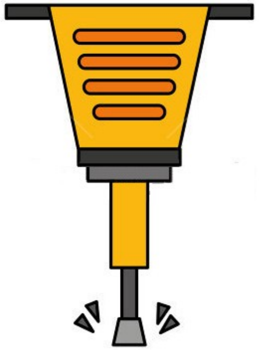
- Automator (cases executor)
  - Launch an application
    - Clean memory state?
      - Fork + copy-on-write
  - Generate input
  - Make the application process the input
  - Detect crashes
  - Kill the application or reset state

Fuzzing



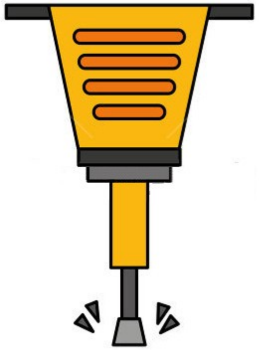
- Automator (cases executor)
  - Performance
    - Minimize I/O
    - Parallel fuzzing (multi-process / multi-core)
  - Multi-platform

Fuzzing



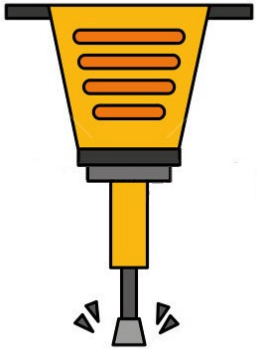
- Automator (cases executor)
  - Reliability
    - Do not leak memory
    - Do not crash
    - It’s going to execute for a long time, unattended
  - Save inputs (or “seeds” that can generate inputs)

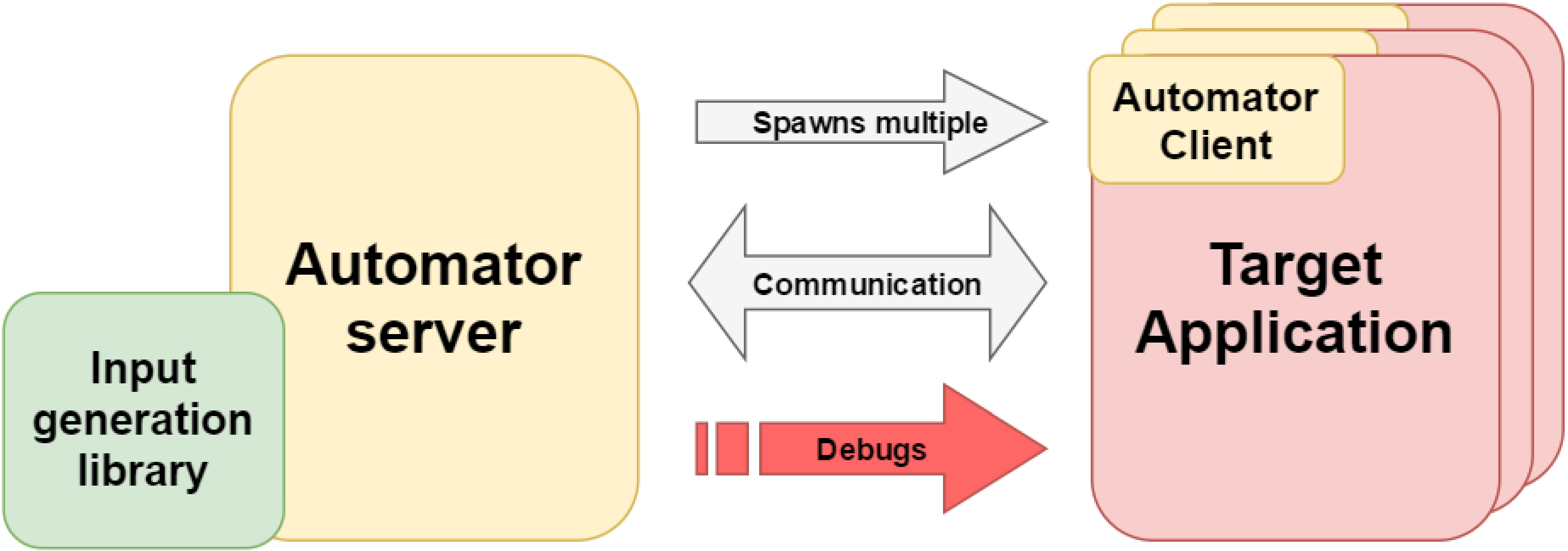
Fuzzing



- Automator (cases executor)
  - Example of an architecture:
    - WebGL/GLSL Fuzzer

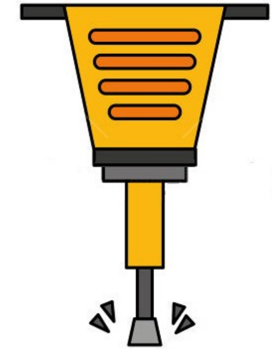
Fuzzing

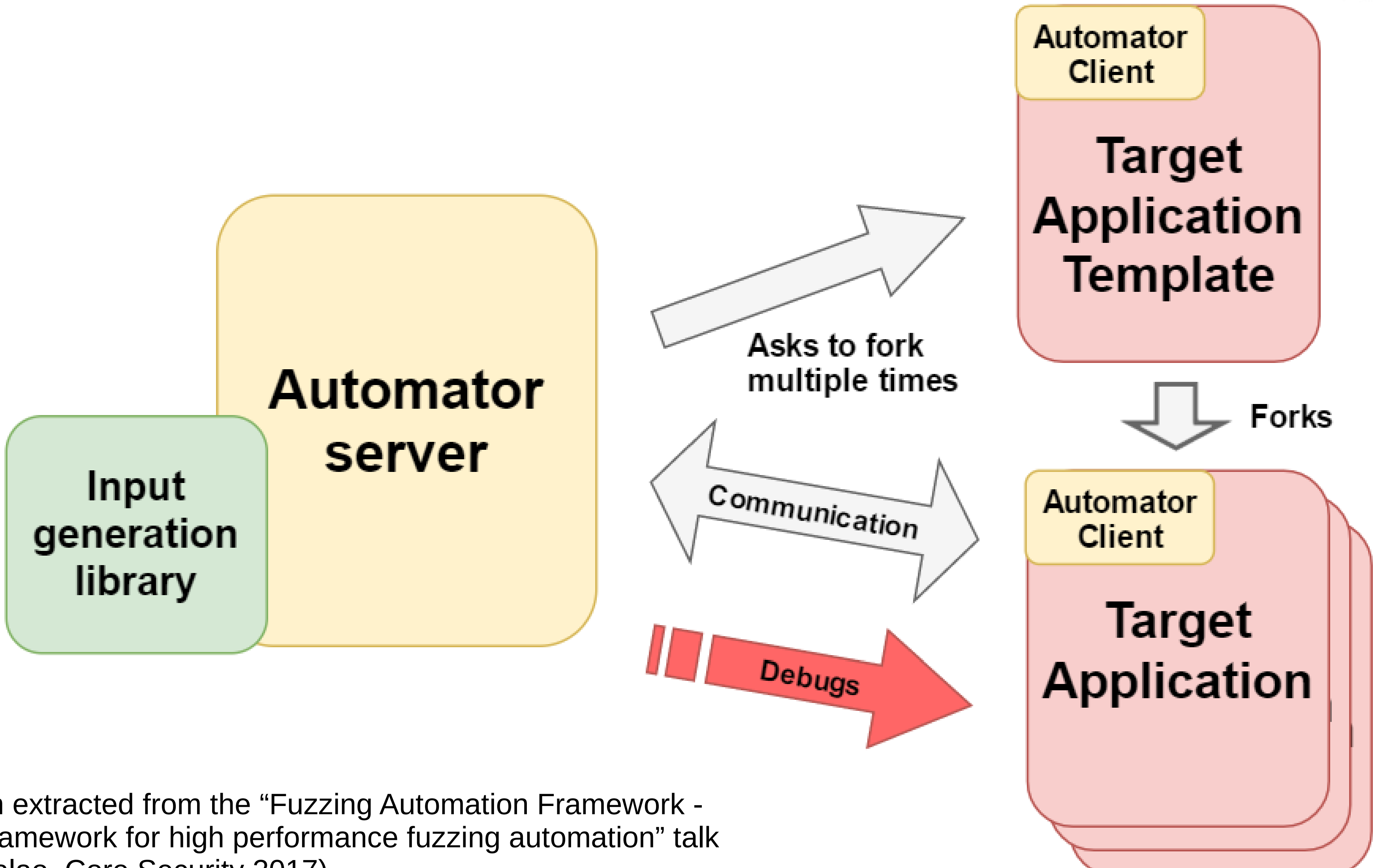




* Diagram extracted from the “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” talk (Martin Balao, Core Security 2017)

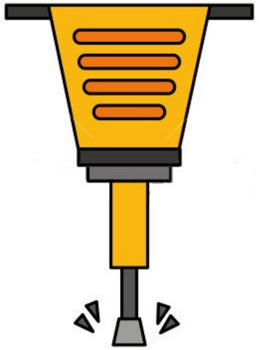
Fuzzing





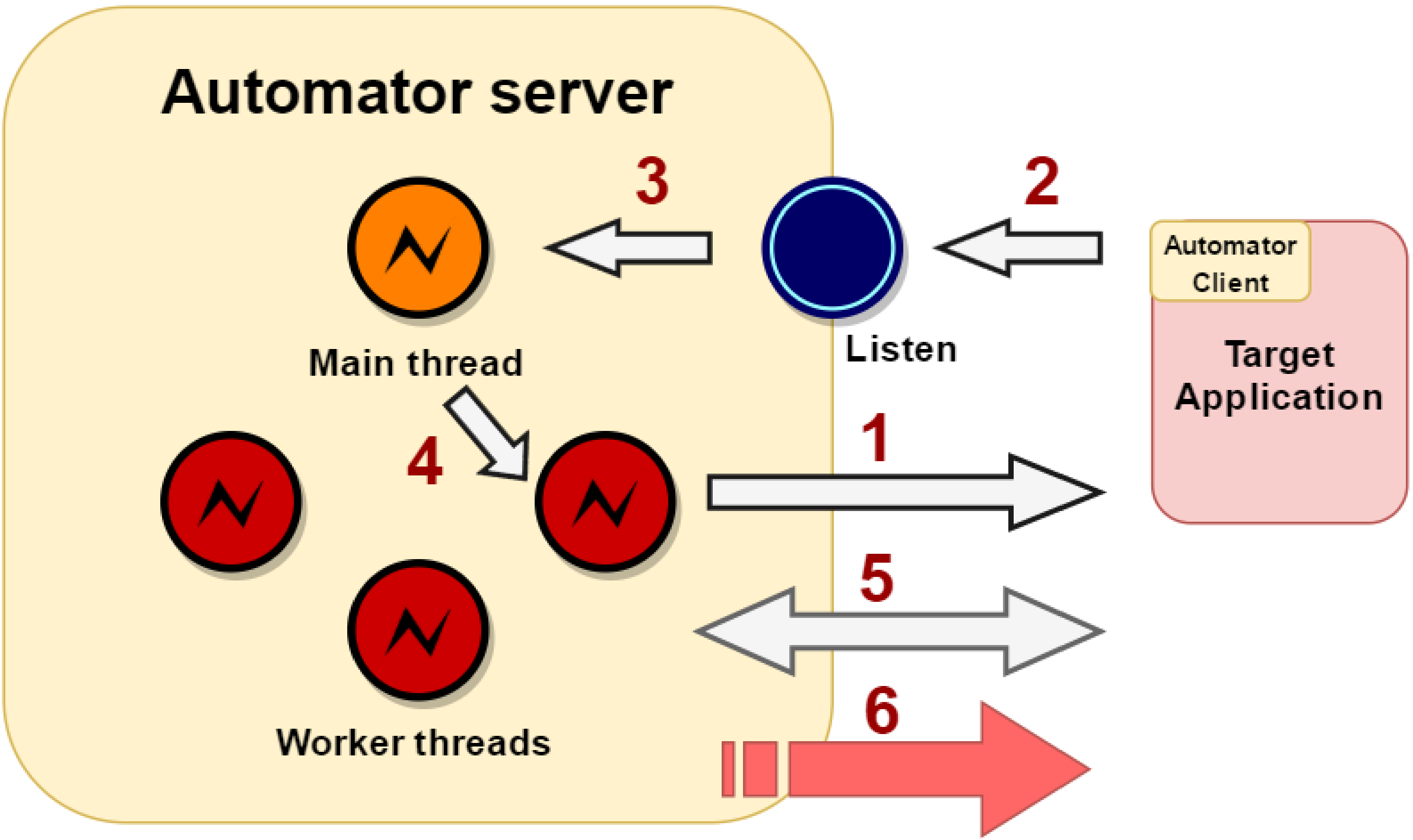
* Diagram extracted from the “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” talk (Martin Balao, Core Security 2017)

Fuzzing



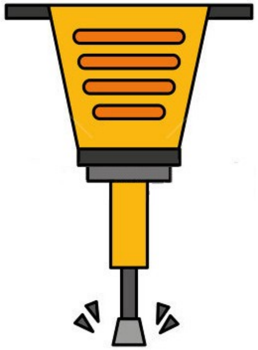
- 1. Each Worker Thread spawns/forks a targeted application
- 2. Targeted application announces its PID
- 3. Main Thread handles the announcement
- 4. Main Thread notifies a Worker Thread about the new application
- 5. A communication is established between the Worker Thread and the targeted application
- 6. Worker Thread debugs the targeted application

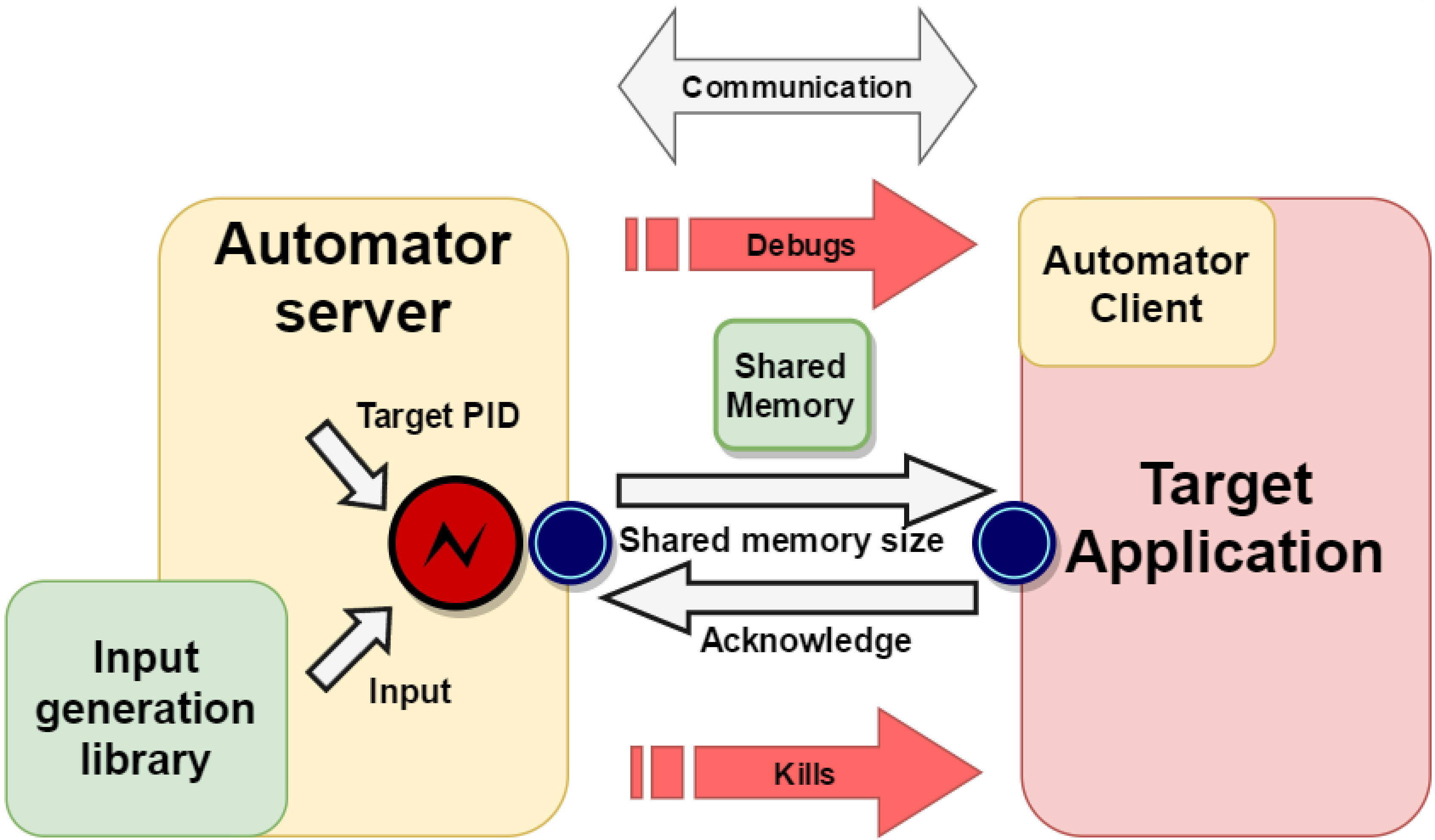
Fuzzing



* Diagram extracted from the “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” talk (Martin Balao, Core Security 2017)

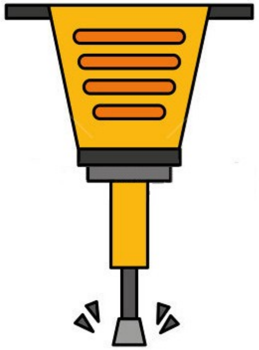
Fuzzing





* Diagram extracted from the “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” talk (Martin Balao, Core Security 2017)

SMT/SAT Solvers



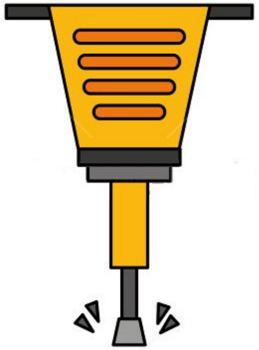
- Resolvers for equation systems
  - SMT (Satisfiability Modulo Theories) solvers take problems in arbitrary forms. Variables can be int. Use SAT solvers as backends

$$x > 4 \wedge (y > -1 \vee x > y + 1)$$

  - SAT solvers take problems in Normal Conjunctive Form (boolean logic). Boolean operands. Variables are true or false

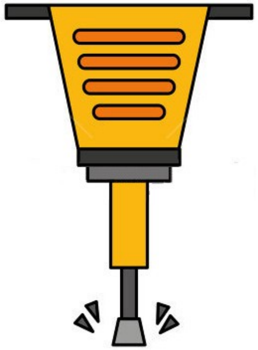
$$\neg A \wedge (B \vee C)$$

SMT/SAT Solvers



- 3 possible states for the solution:
  - Cannot be satisfied
  - Can be satisfied (and one or more solution cases)
  - Don’t know! Timeout?
- Not new, but computing power now made possible to solve problems that some time ago were not
- Has application to an infinite number of problems
- z3 is a library that has SMT/SAT solvers. Developed in C++ but has bindings for multiple languages (Python, .NET, Java, etc.)

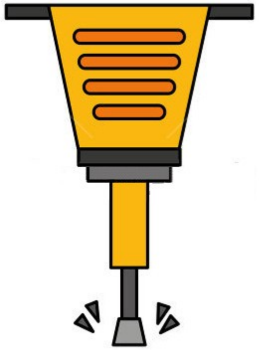
SMT/SAT Solvers



- How can we solve this equations system?

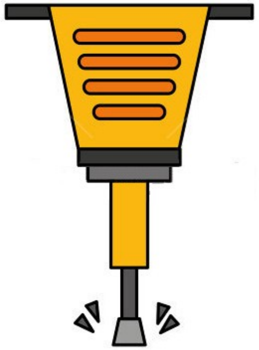
$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - z &= 0 \end{aligned}$$

SMT/SAT Solvers



```python
#!/usr/bin/python
from z3 import *
x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print s.check()
print s.model()
```

sat
[z = -2, y = -2, x = 1]

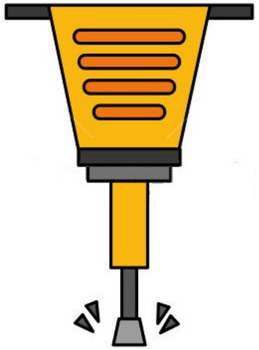


SMT/SAT Solvers

- How can we solve this Sudoku?

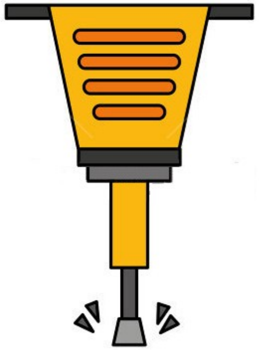
		5	3					
8							2	
	7			1		5		
4					5	3		
	1			7				6
		3	2				8	
	6		5					9
		4					3	
					9	7		

SMT/SAT Solvers



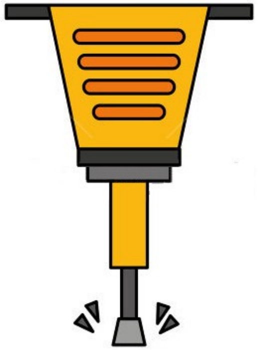
- Cells in the board have to be filled with numbers from 1 to 9
- Numbers cannot be repeated:
  - Per row
  - Per column
  - Per sub-quadrant
- Can we model this problem so it can be adequate for an SMT solver?
  How can we model constraints?

SMT/SAT Solvers



- Model the board as an Int matrix ([][]): cells=[[Int('cell%d%d' % (r, c)) for c in range(9)] for r in range(9)]
- Add constraints for cells that already have an assigned value: s.add(cells[current_row][current_column]==int(i))
- Add constraints to each cell for the solution to be between 1 and 9: s.add(cells[r][c]>=1), s.add(cells[r][c]<=9)

SMT/SAT Solvers

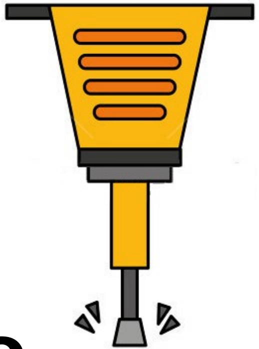


- Add constraints for column and row uniqueness: s.add(Distinct(cells[r][0],… cells[r][8])) y s.add(Distinct(cells[0][c],… cells[8][c]))
- Add constraints for sub-quadrant uniqueness: s.add(Distinct(cells[r+0][c+0]…))
- Check if there is a solution: s.check()
- Obtain a solution: m=s.model()

SMT/SAT Solvers

- How can we solve this minesweeper?

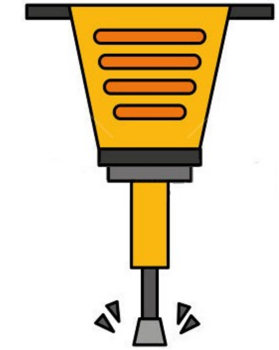
SMT/SAT Solvers



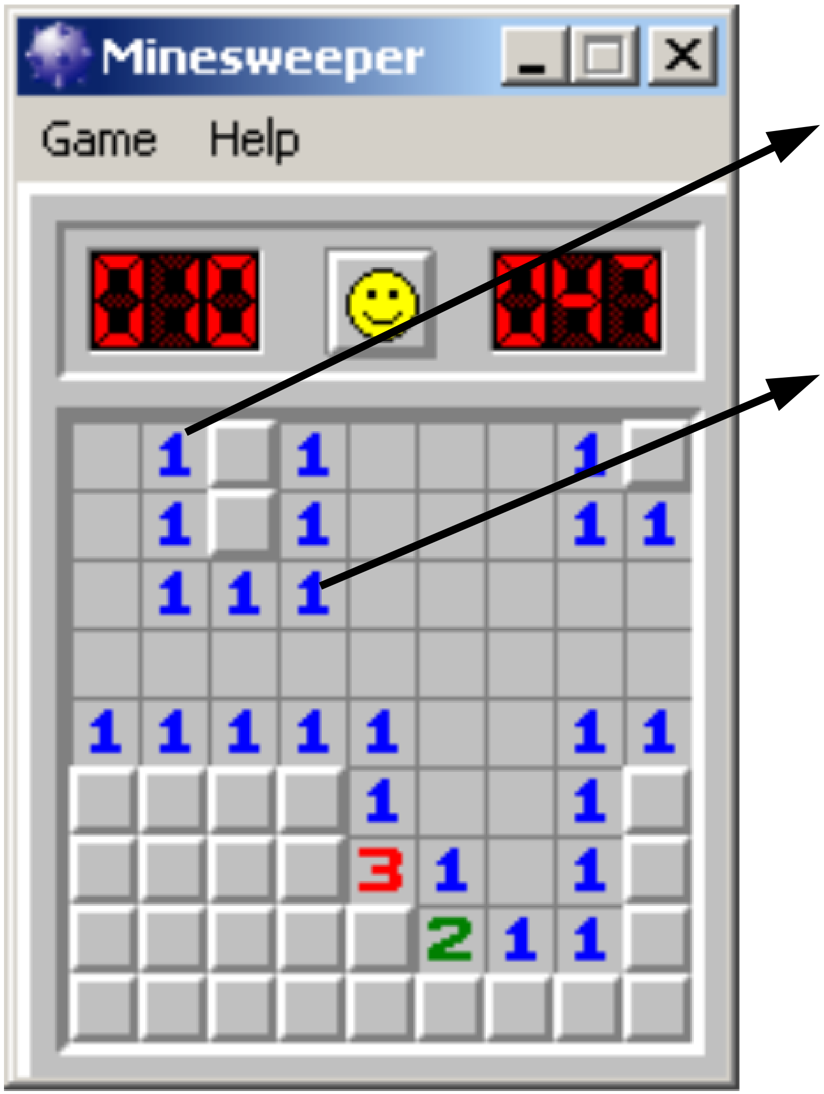
1) Is it safe to tap here?

2) and here?

Assume that there is a mine in each place, can constraints imposed by nearby cells be satisfied?



SMT/SAT Solvers

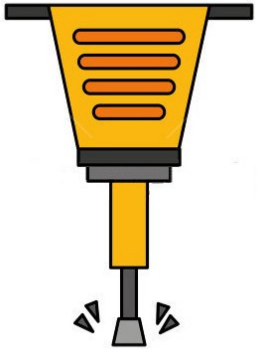


This 1 imposes the following condition: 1) + 2) = 1

This 1 imposes the following condition: 2) = 1

If we assume that the mine is in 1), the following condition is added: 1) = 1

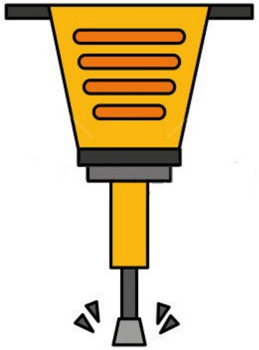
SMT/SAT Solvers



SMT solver returns that the equations system has no solution. Thus, mine is not in 1)

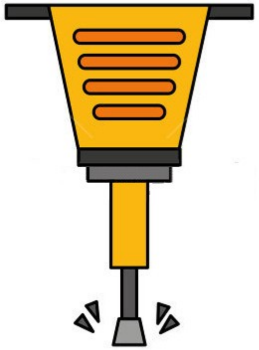
If there is at least 1 solution, **we cannot decide** whether there is a mine or not

SMT/SAT Solvers



- It’s important to correctly model the problem and make the question in a way that the SMT solver can answer it (within a reasonable time frame)
- It’s also possible to resolve optimization problems in z3

SMT/SAT Solvers

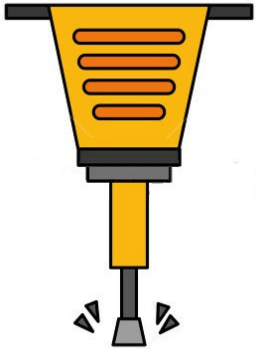


- Cracking a cipher text (plain text XOR key) with z3

Inputs		Outputs
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

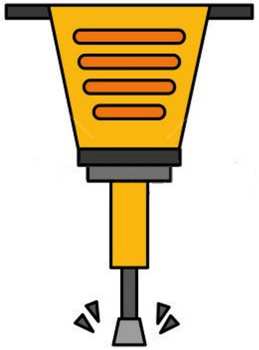
XOR Truth Table

SMT/SAT Solvers



- Let’s assume that plain text is a text in English. Key length is unknown, but much smaller than cipher text
- One approach is to try different key lengths and for each one maximize the number of alphabetical characters
- We need to add XOR operation and periodic key repetition constraints. I.e. if key has a length of 5, byte 0 of the key will be XORed with cipher text in positions multiple of 5

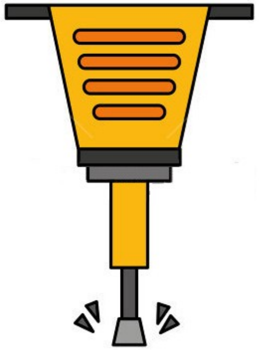
SMT/SAT Solvers



- Variables to model the problem

```
# variables for each byte of key:
key=[BitVec('key_%d' % i, 8) for i in range (KEY_LEN)]

# variables for each byte of input cipher text:
cipher=[BitVec('cipher_%d' % i, 8) for i in range (cipher_len)]

# variables for each byte of input plain text:
plain=[BitVec('plain_%d' % i, 8) for i in range (cipher_len)]

# variable for each byte of plain text: 1 if the byte in 'a'...'z' range:
az_in_plain=[Int('az_in_plain_%d' % i) for i in range (cipher_len)]
```

SMT/SAT Solvers



- Variables to model the problem

Example (key length = 5)

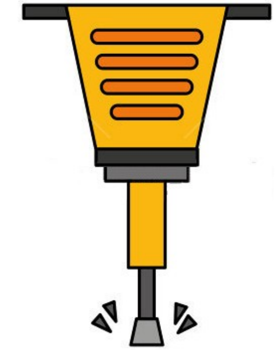
BitVec (8 bits)

- **Key = [0x55, 0x03, 0xAB, 0x1C, 0xE5]**
- **cipher text = [0x34, 0x61, 0x54, 0x7F, 0x81, ...]**
- **plain text = [0x61, 0x62, 0xFF, 0x63, 0x64, ...]**
- **az_in_plain= [1, 1, 0, 1, 1, ...]**

Int

We want to maximize the sum of az_in_plain

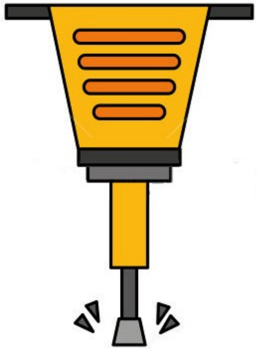
SMT/SAT Solvers



- Problem constraints

```
for i in range(cipher_len):
   # assign each byte of cipher text from the input file:
   s.add(cipher[i]==ord(cipher_file[i]))
   # plain text is cipher text XOR-ed with key:
   s.add(plain[i]==cipher[i]^key[i % KEY_LEN])
   # each byte must be in printable range, or CR of LF:
   s.add(Or(And(plain[i]>=0x20,
plain[i]<=0x7E),plain[i]==0xA,plain[i]==0xD))
   # 1 if in 'a'...'z' range, 0 otherwise:

s.add(az_in_plain[i]==If(And(plain[i]>=ord('a'),plain[i]<=ord('
z')), 1, 0))
```

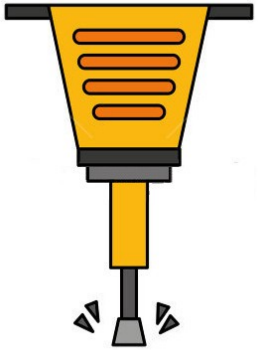
SMT/SAT Solvers



- Solution

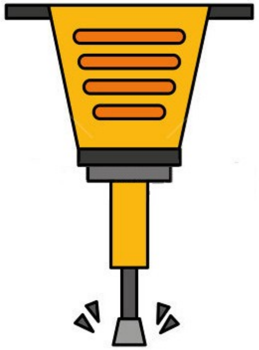
```
s=Optimize()

s.maximize(Sum(*az_in_plain))
if s.check()==unsat:
    return
m=s.model()

test_key="".join(chr(int(obj_to_string(m[key[i]]))) for i in
range(KEY_LEN))
```

SMT/SAT Solvers



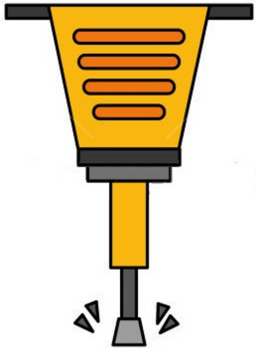
- Solution
  - Multiple variables can be optimized at the same time
  - It’s possible to assume that the appearance of certain letters together is more likely and use this information as an optimization vector
  - It’s possible to weigh optimization vectors and “educate” the search for solutions

SMT/SAT Solvers



- How do SMT/SAT solvers work?
  - Common theories
    - Bit Vectors
      - Ideal to represent finite range data types. I.e. 32 bits integers. This enables to model “overflows” and “underflows”
    - Arrays
      - Variable length
    - Integers
    - Not-interpreted functions
      - Given the same inputs, the same output is returned

SMT/SAT Solvers

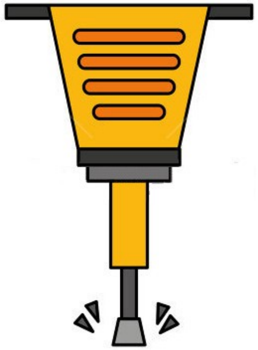


- How do SMT/SAT solvers work?
  - Base of constraints in normal conjunctive form (every boolean formula can be expressed in this form)

$$x_1 \vee x_2 \vee x_3$$

  - SAT solver assigns a truth value to one variable, and start making deductions based on that

SMT/SAT Solvers



- How do SMT/SAT solvers work?

$$x_1 = true$$

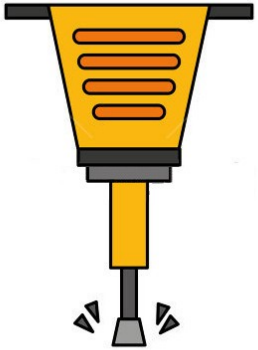
$$-x_1 \vee x_7 \Rightarrow x_7 = true$$

$$-x_7 \vee x_5 \vee -x_1 \Rightarrow x_5 = true$$

...

  - It may either assign a value to each variable without violating constraints or come to a contradiction. If it comes to a contradiction, it has to summarize it in a single clause and add it to the base of constraints to avoid it next time

SMT/SAT Solvers



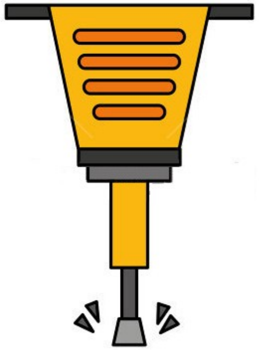
- How do SMT/SAT solvers work?

$$x>5 \land y<5 \land (y>x \lor y>2)$$

  - Part of this formula requires reasoning in a specific domain (i.e. set of integers) and the other part is boolean logic that can be expressed in normal conjunctive form (SAT solver)

$$F1 \land F2 \land (F3 \lor F4)$$

SMT/SAT Solvers



- How do SMT/SAT solvers work?

$$F1 \wedge F2 \wedge (F3 \vee F4)$$

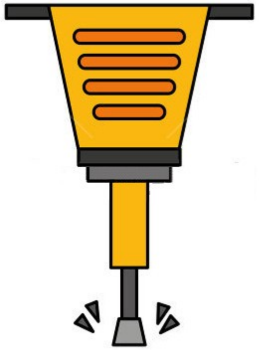
↓

SAT SOLVER

↓

$$F1 = true, F2 = true, F3 = true$$

SMT/SAT Solvers



- How do SMT/SAT solvers work?

$$x > 5, y < 5, y > x$$

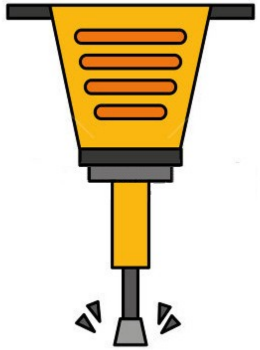
↓

Theory Solver (linear arithmetic)

↓

NO

SMT/SAT Solvers



- How do SMT/SAT solvers work?

$$F1 \wedge F2 \wedge (F3 \vee F4)$$

$$\neg (F1 \wedge F2 \wedge F3)$$

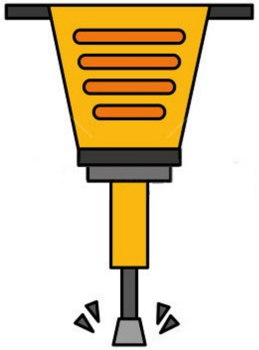
↓

SAT SOLVER

↓

$$F1 = true, F2 = true, F4 = true$$

SMT/SAT Solvers



- How do SMT/SAT solvers work?

$$x > 5, y < 5, y > 2$$

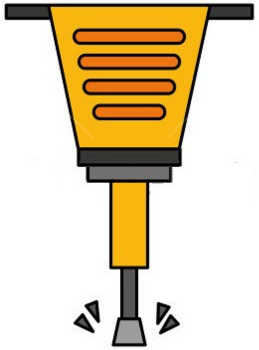
↓

Theory Solver (linear arithmetic)

↓

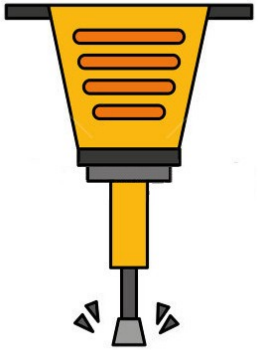
Yes $x = 6, y = 3$ ✓

Symbolic Execution



- How can SMT/SAT solvers contribute to vulnerability finding in source code?
  - Symbolic execution
    - Technique to analyze programs
    - How is the behavior going to be in a potentially infinite input set?
    - Improve code coverage
    - When a problem is found, it can provide a set of inputs to reproduce it (as opposed to static analysis)

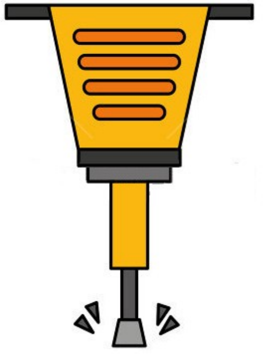
Symbolic Execution



```
void foo ( int x, int y) {
    int t = 0;

    if (x > y) {
        t = x;
    } else {
        t = y;
    }

    if (t < x) {
        assert false;
    }

}
```

Are there a pair of x, y inputs that trigger the assertion?

Symbolic Execution

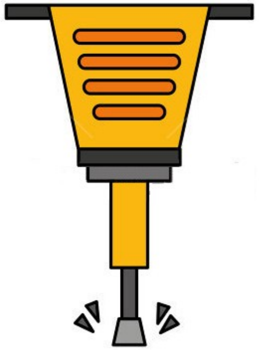


Program state characterization: 3 state variables

x	y	t
4	4	0
4	4	4

Assertion is not triggered: x == t

Symbolic Execution

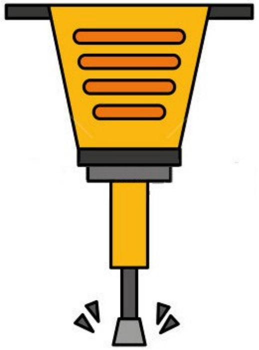


x	y	t
2	1	0
2	1	2

Assertion is not triggered: x == t

But, how can we make sure that there are no inputs for which the assertion is triggered?

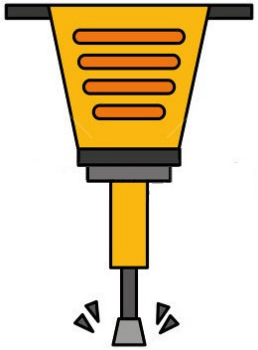
Symbolic Execution



- Program state redefinition, mapping unknown variables (x, y) to symbolic values (χ, γ)

x	y	t
χ	γ	0
χ	γ	t_0

$$\overbrace{(x>y)\Rightarrow x,(x\leq y)\Rightarrow y}^{t0}$$



Symbolic Execution

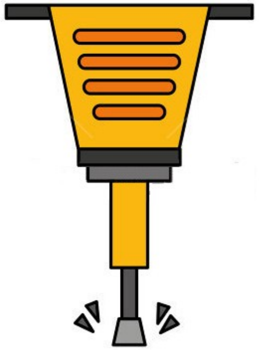
- Is it possible to satisfy the following constraints? Is there a solution for this equations system?

$$t_0 < x$$

$$(x > y) \Rightarrow t_0 = x$$

$$(x \leq y) \Rightarrow t_0 = y$$

Symbolic Execution



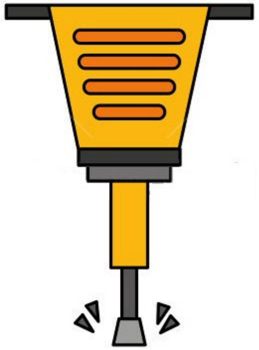
- Is it possible to satisfy the following constraints? Is there a solution for this equations system?

$$t_0 < x$$

$$(x > y) \Rightarrow t_0 = x$$ ❌

$$(x \leq y) \Rightarrow t_0 = y$$

Symbolic Execution



- Is it possible to satisfy the following constraints? Is there a solution for this equations system?

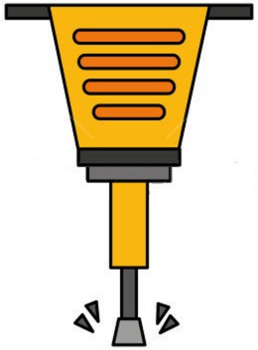
$$t_0 < x$$

$$(x > y) \Rightarrow t_0 = x$$

$$(x \leq y) \Rightarrow t_0 = y \quad \Rightarrow \quad (t_0 < x \leq y = t_0)$$

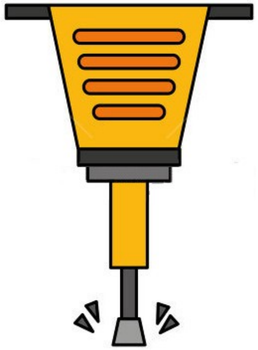
$$(t_0 < t_0)$$

Symbolic Execution



- Is it possible to satisfy the following constraints? Is there a solution for this equations system?
  - An SMT/SAT solver can bring the answer!
  - In general, despite there can be many variables involved in a real problem, there aren’t so many degrees of freedom: variables tend to be conditioned by others
    - Depends on the size of the unit that is being analyzed
    - If a function is simple, all paths can be analyzed at once

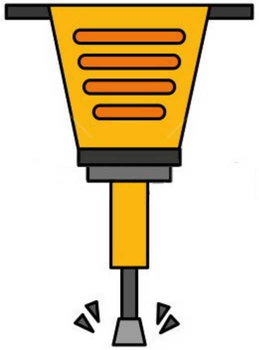
Symbolic Execution

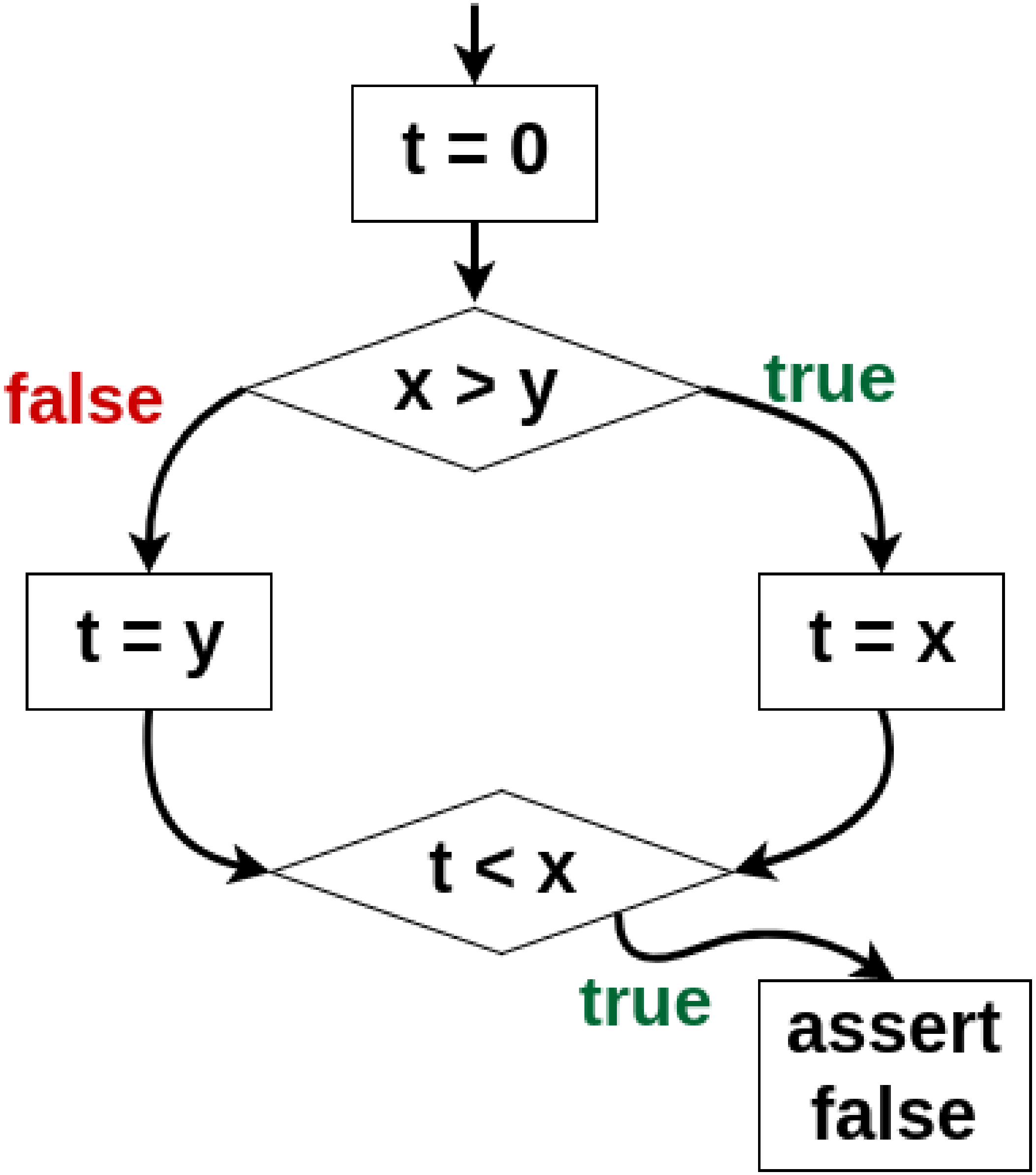


```
#!/usr/bin/python
from z3 import *

x = Int('x')
y = Int('y')
t = Int('t')
s = Solver()

s.add(t < x)
s.add(If(x > y, t == x, t == y))

print s.check()
print s.model()
```

unsat

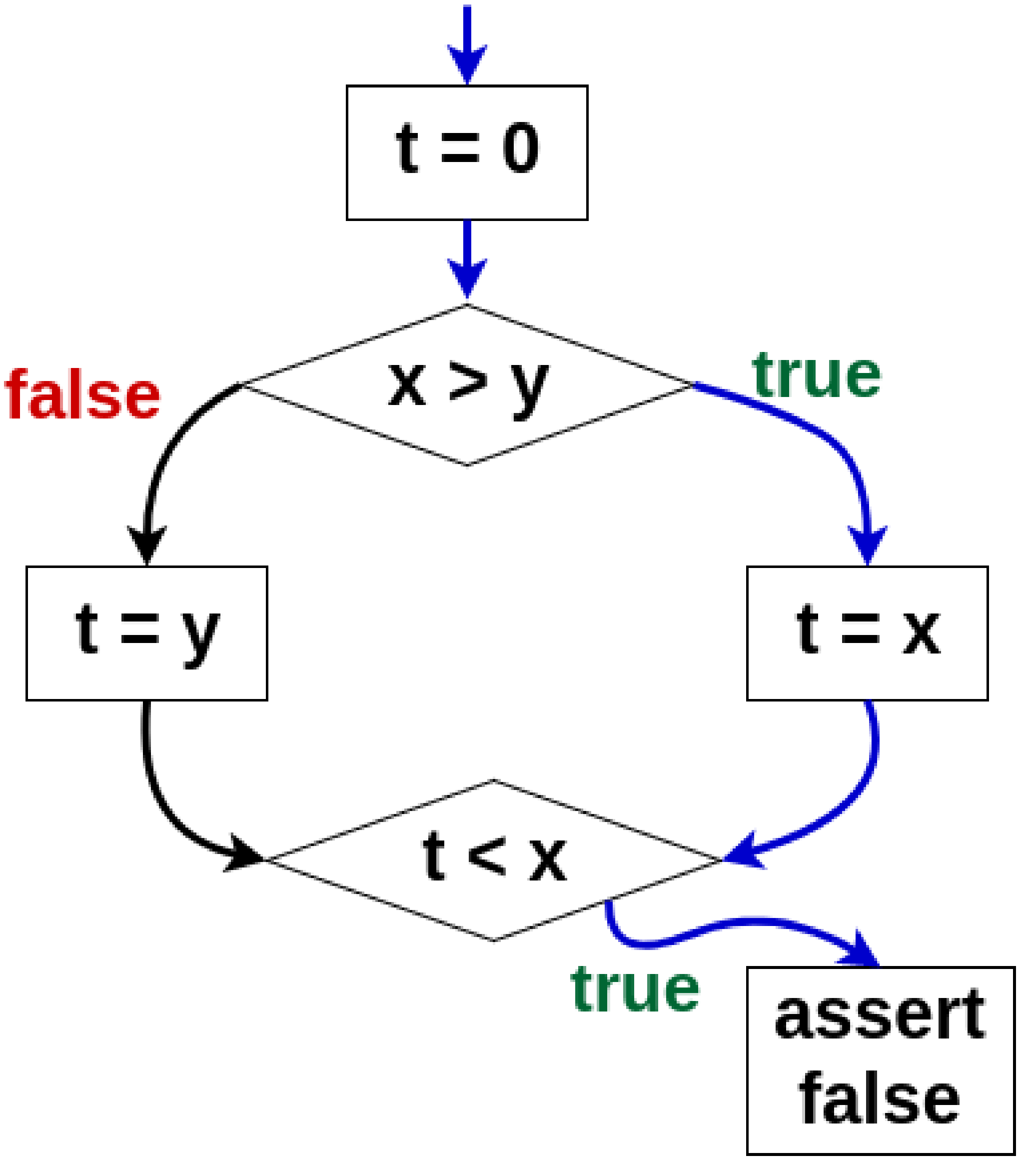
Symbolic Execution





If software being analyzed is too complex, path exploration can be used

Symbolic Execution



Constraints:

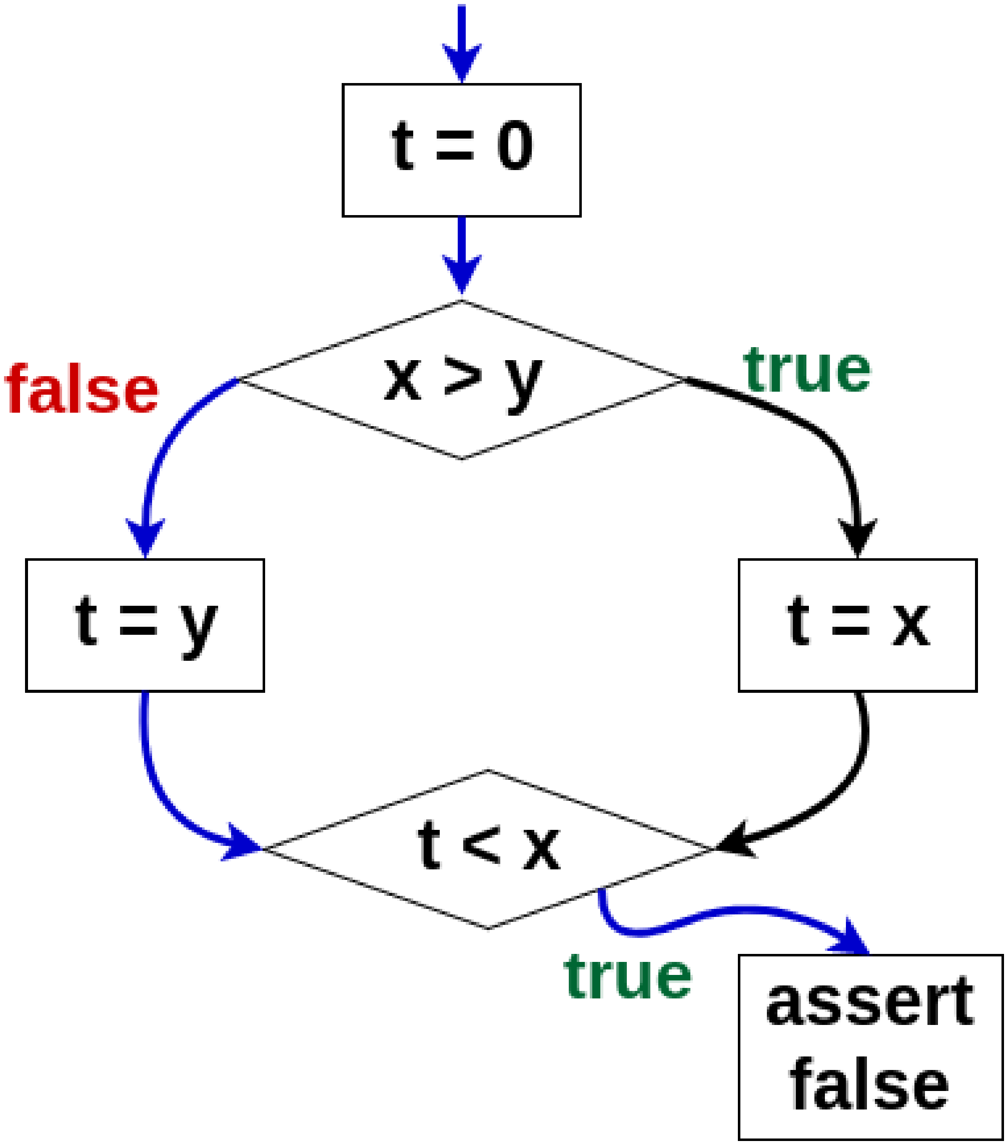
$$t_0 = x$$

$$t_0 < x$$

Simpler equations system when exploring only 1 path

Question is just if this path is feasible

Symbolic Execution



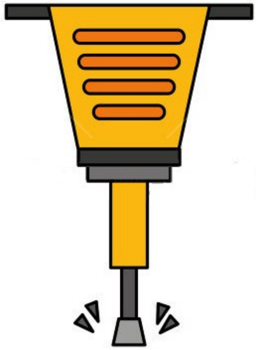
Constraints:

$$x \leq y = t_0$$

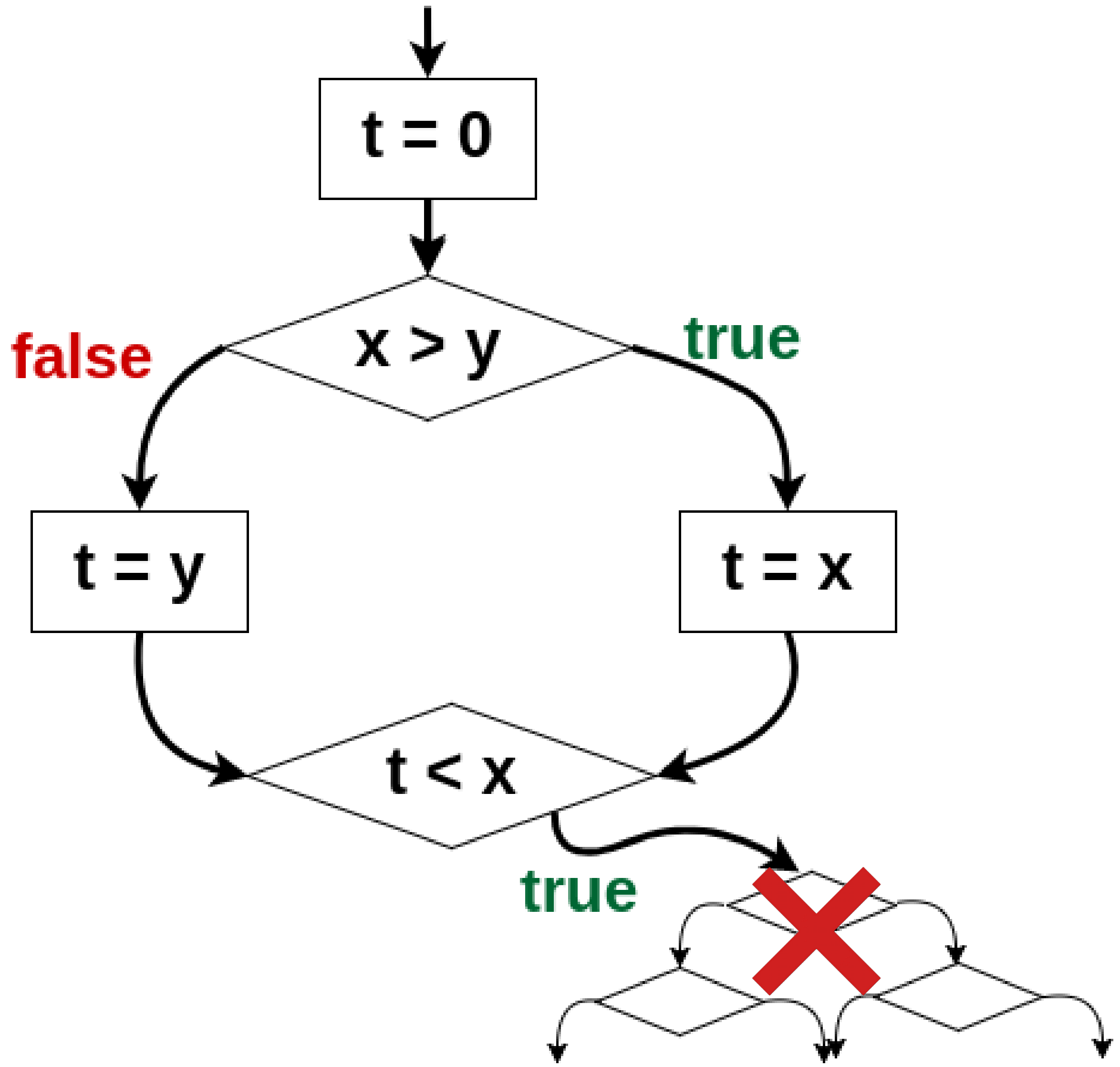
$$t_0 < x$$

Simpler equations system when exploring only 1 path

Question is just if this path is feasible

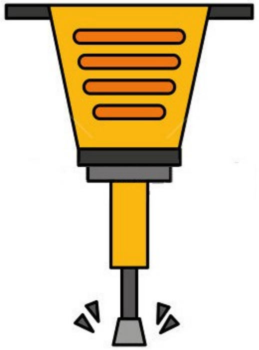


Symbolic Execution



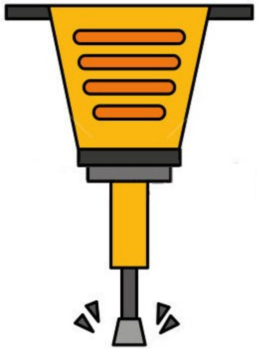
More paths are explored but each of them is simpler. It's possible to use strategies to discard unfeasible paths.

Symbolic Execution



- Symbolic execution can be used as a complement to real execution (fuzzing / testing). I.e:
  - A code-coverage tool shows that a program path was not executed doing fuzzing
  - We take a close case (generated with real input) and apply symbolic execution from a known state to trigger non-executed paths

Lab



Lab 6.1: Implement “generate_input” function in fuzzer.py to crash main, without doing reverse engineering on the binary

- In case of not crashing it, do reverse engineering to guide automated inputs generation
- In case of not crashing it, analyze the source code to guide automated inputs generation

References

- Fuzzing Brute Force Vulnerability Discovery
- Examples obtained from:
  - “Quick introduction into SAT/SMT solvers and symbolic execution” - Dennis Yurichev
  - MITOpenCourseware – Computer System Security – Lecture 10: Symbolic Execution – Armando Solar-Lezama