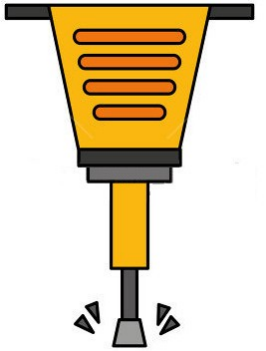
Ingeniería Inversa
Clase 6

Fuzzing

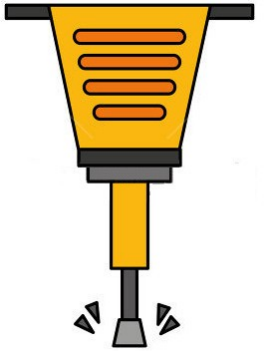
martin.uy
Open by default.

Fuzzing



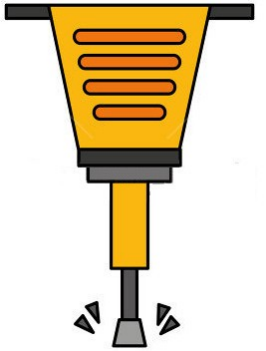
- Testing de caja gris
  - No es necesario acceso al código fuente. Si hubiera acceso, puede ser útil pero no es necesario entenderlo completamente
  - Se puede guiar haciendo ingeniería inversa
- Enviar, de forma automatizada, inputs válidos e inválidos a una aplicación con el objetivo de triggerear mal comportamiento
  - Eventualmente, problemas de seguridad
- Buscar vulnerabilidades (bug hunting)
  - Interno
  - Externo (bug bounty, security advisory, research)

Fuzzing



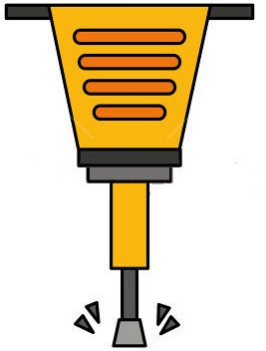
- Aplicable a todo tipo de inputs:
  - Aplicaciones web
    - Fuzzing de parámetros POST/GET
  - Formatos de archivos (doc, jpg, mp3, etc.) y sistemas de archivos
    - Vulnerabilidades en el parser
  - Protocolos de red
  - Lenguajes de programación
    - Ej. JavaScript puede ser visto como un input complejo para un browser
  - Drivers
    - Ej. *ioctls* atendidas por un driver, filtros de sistemas de archivos/red, operaciones de read/write en un dispositivo de caracteres, etc.
  - Etc.

Fuzzing



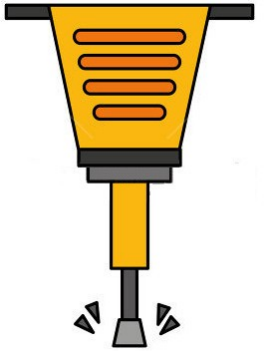
- Importancia del fuzzing
  - Disciplina relativamente nueva
  - Esfuerzo significativo de la industria
    - ClusterFuzz, OSS-Fuzz (Google)
    - SAGE (Microsoft)
  - Todavía resta mucho por hacer
  - Relevante por la cantidad de vulnerabilidades encontradas

Fuzzing



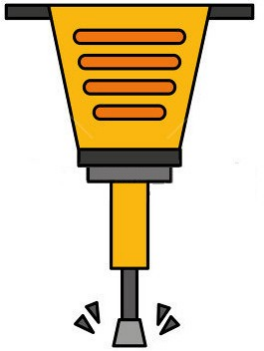
- Importancia del fuzzing
  - Fuzzer comerciales y libres
    - PeachFuzzer (comercial)
    - SPIKE (libre)
    - AFL (libre)
  - Frameworks de fuzzing genéricos
  - Fuzzers personalizados (ad-hoc)

Fuzzing



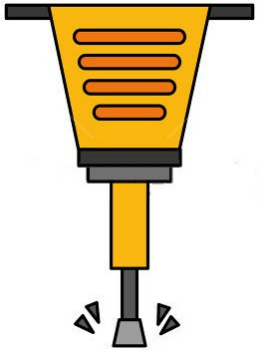
- Límites del fuzzing
  - Bugs lógicos o ataques sobre datos
    - El fuzzer no se enfoca generalmente en bugs lógicos como ser information-disclousure o escalación de privilegios
  - Corrupciones de memoria que no generan crashes
    - Es necesario recompilar con librerías (o flags de compilación) que ponen sentinels alrededor de los buffers para exhibir corrupciones de memoria
  - Condiciones de carrera
    - Bugs difíciles de reproducir

Fuzzing



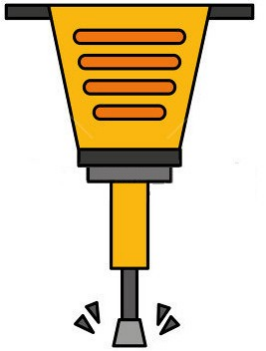
- Tipos de fuzzers
  - Fuzzers puramente randómicos
    - Generan inputs con basura
    - Costo nulo pero muy tontos
  - Mutacionales
    - Inputs válidos son alterados randómicamente (ej. mutaciones, permutaciones, sustituciones con diccionarios o números mágicos)
    - Es importante un set de inputs representativo

Fuzzing



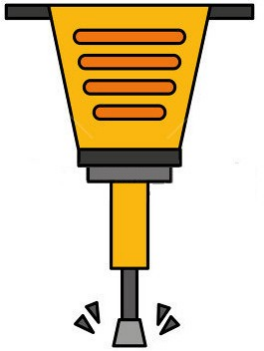
- Tipos de fuzzers
  - Evolutivos o Genéticos
    - Variante de los fuzzers mutacionales, se retroalimentan con métricas y la generación es guiada
  - Generacionales
    - Los inputs se generan en base a un modelo o especificación (ej. gramática de un lenguaje o de un protocolo de comunicaciones)
    - Alto costo de desarrollo. No siempre la especificación está disponible. Puede ser necesario hacer ingeniería inversa
  - Mixtos

Fuzzing



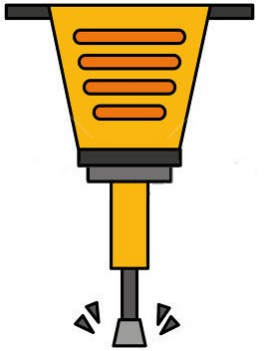
- Métricas
  - Ejercitar la mayor cantidad posible de flujos de ejecución y estados de la memoria
    - *Code-coverage*
  - Performance
  - Detección confiable de crashes
  - Casos reproducibles (documentados)

Fuzzing



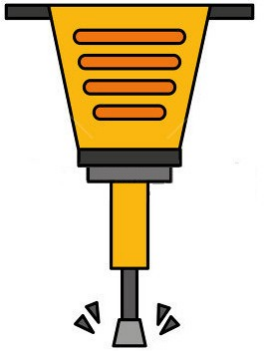
- Fases
  - Identificación y análisis de formato para los inputs
    - No siempre son claros:
      - ¿Sockets?
      - ¿Syscalls?
      - ¿Archivos? ¿Meta-información?
      - ¿Variables de entorno? ¿Cuáles?
      - ¿Registry? ¿Cuál clave?
      - ¿Mecanismos de IPC?

Fuzzing



- Fases
  - Generación automática y performante de inputs
  - Automatización
    - Envío performante de inputs
    - Detección confiable de crashes
  - Análisis de los crashes
    - Minimización de los inputs que generan el crash (manual o automatizado)
    - Análisis de explotabilidad (manual)

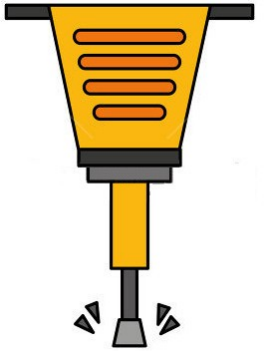
Fuzzing



- Problema de los fuzzers puramente randómicos

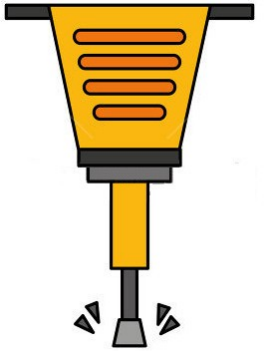
Demo 6.1

Fuzzing



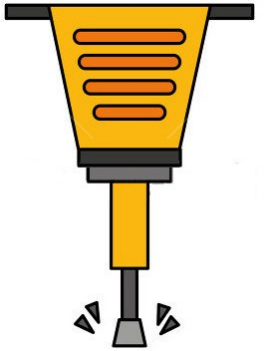
- Análisis del formato de los inputs
  - Campos clave-valor (ej. JSON, HTTP header)
  - Campos de largo variable
  - Campos delimitados por caracteres especiales
  - Inputs de texto (ASCII, UTF-8) o inputs binarios
  - Entender el formato de los inputs puede ayudar a enfocar mejor el esfuerzo. En ocasiones, analizar los inputs requiere hacer ingeniería inversa

Fuzzing



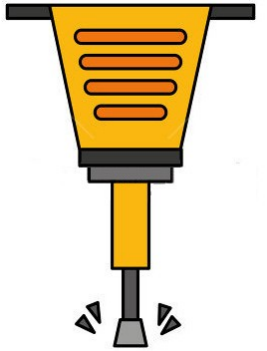
- Supongamos que una aplicación recibe un entero de 64 bits como input
  - Probar el rango entero tiene un costo de tiempo y cómputo excesivo
  - ¿Podemos hacer que el fuzzer sea más inteligente? ¿Que heurísticas podemos aplicar a este caso?

Fuzzing



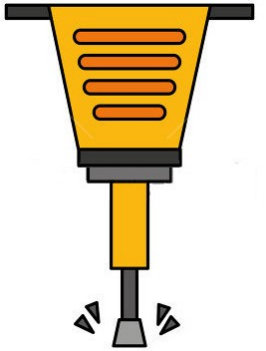
- Extremos del rango, asumiendo diferentes tamaños para representar el entero:
  - 0...0xFF, 0...0xFF, 0...0xFFFF, 0...0xFFFFFFFF, 0…0xFFFFFFFFFFFFFFFF
  - ¿Qué sucedería si el entero recibido es sumado a alguna constante? (ej. para alocación de memoria)
  - Probar valores cercanos a los extremos:
    - 0, 1, 2, 3, 4 … 0xFD, 0xFE, 0xFF, etc.

Fuzzing



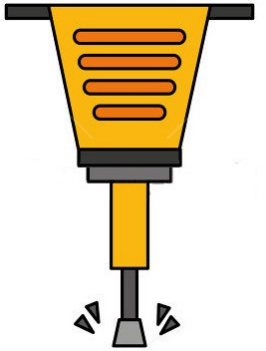
- ¿Qué sucede si el entero es multiplicado por una constante? (ej. 2)
  - Probar extremos del rango divididos por la constante y sus cercanías. Ej.: 0xFF/2, 0xFFFF/2, 0xFFFFFFFF/2, etc.
- Probar magic numbers
  - Enteros que puedan tener un significado especial para el contexto (ej. constantes, valores de un enumerativo)

Fuzzing



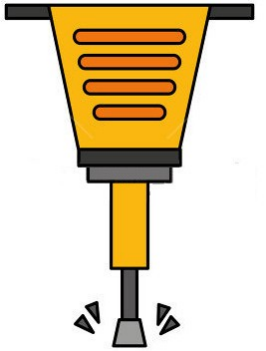
- Supongamos que una aplicación recibe un string como input
  - ¿Que heurísticas podemos aplicar aquí?

Fuzzing



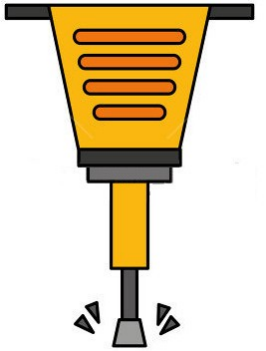
- Diferentes encodings y caracteres multi-byte
  - ASCII, UTF-8, UTF-16, UTF-32, html encoding, etc.
  - ¿Hay conversiones de formato? ¿Están bien implementadas? ¿Hay problemas al calcular los largos?
- Caracteres de escape, delimitadores, caracteres especiales según el contexto. Ej. si estamos probando un parser de XML probablemente nos interese probar caracteres como “<” y secuencias como “<![CDATA[]]>”.
- ¿Strings terminados en NULL? ¿El tipo de dato del string tiene un largo al comienzo? (ej. BSTR)
- Repetición de caracteres delimitadores (¿es posible triggerear un integer overflow en alguna variable?)
- Diferentes largos
- Format strings (“%s, %d ...”)
- Palabras de un diccionario (según el contexto)

Fuzzing



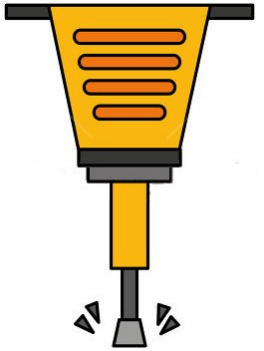
- Fuzzing en memoria
  - Los inputs son inyectados directamente en la memoria del proceso targeteado
    - ¿Cómo lo hacemos?

Fuzzing



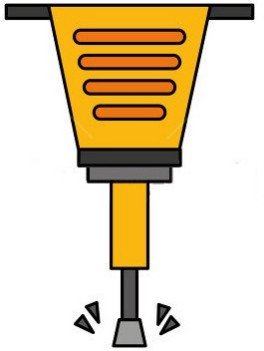
- Fuzzing en memoria
  - Mejorar performance
  - Evitar post-procesamiento del input generado
    - Cifrar, firmar, calcular checksums, incluir token previo u otros controles de integridad, etc.
  - Evitar estados previos en la máquina de estados
    - Ej. autenticación

Fuzzing



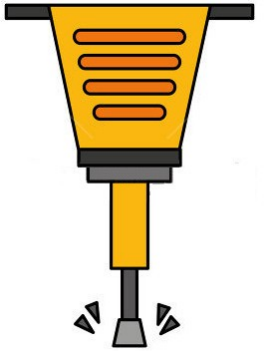
- Fuzzing en memoria
  - Mayor costo de implementación
  - Debemos asegurarnos de partir de un estado de la memoria válido (al que podemos llegar a partir de una cierta secuencia de inputs)
    - Esto no elimina falsos positivos. Ej. un filtro previo puede no permitir el input que genera el crash

Fuzzing



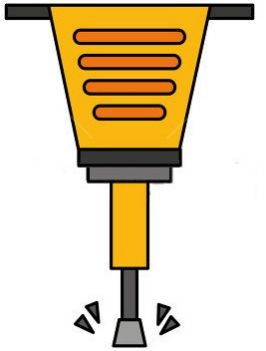
- Fuzzing en memoria
  - Patchear memoria del proceso para ejecutar trampolines (hooks)
    - ¿Cómo?
  - Frameworks de instrumentación binaria
    - DynamoRIO
    - PIN
  - Recompilar con hooks (si el código fuente está disponible)

Fuzzing



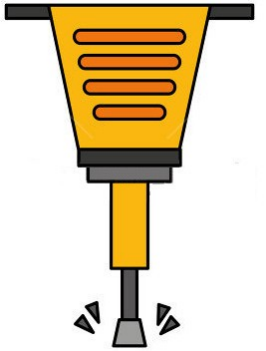
- Automatización
  - La automatización es todo
  - El costo computacional es bajo en relación al costo del talento calificado
  - La cantidad de casos que se puede probar por unidad de tiempo es significativamente mayor, y los casos se pueden utilizar en múltiples targets
  - Enfocar el esfuerzo en una buena generación y ejecución de casos

Fuzzing



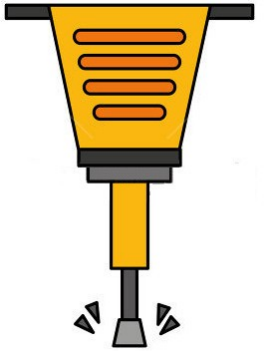
- Automatizador (ejecutor de casos)
  - Lanzar aplicación
    - ¿Estado de la memoria limpio?
      - Fork + copy-on-write
  - Generar input
  - Hacer que la aplicación procese el input
  - Detectar crashes
  - Matar la aplicación o resetear estado

Fuzzing



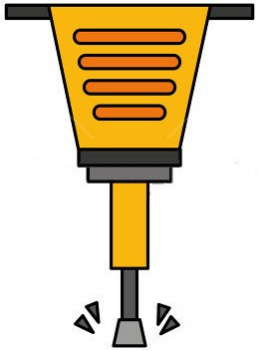
- Automatizador (ejecutor de casos)
  - Desempeño
    - Minimizar I/O
    - Fuzzing paralelo (multi-proceso / multi-core)
  - Multi-plataforma

Fuzzing



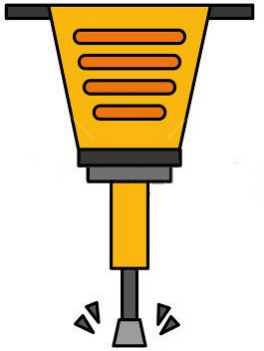
- Automatizador (ejecutor de casos)
  - Confiabilidad
    - No leakear memoria
    - No crashear
    - Va a ejecutar por mucho tiempo, desatendido
  - Guardar inputs (o “seeds” que generen inputs)

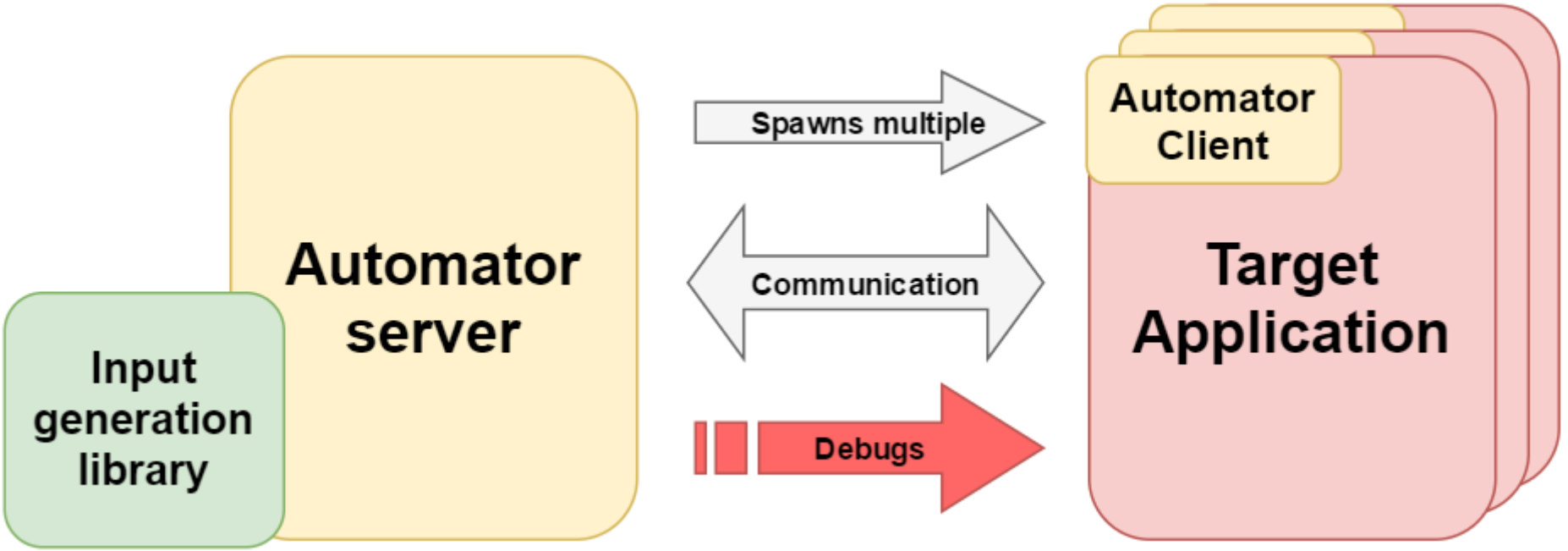
Fuzzing



- Automatizador (ejecutor de casos)
  - Arquitectura de ejemplo:
    - WebGL/GLSL Fuzzer

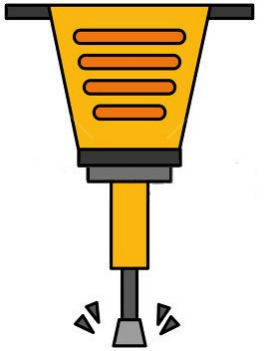
Fuzzing

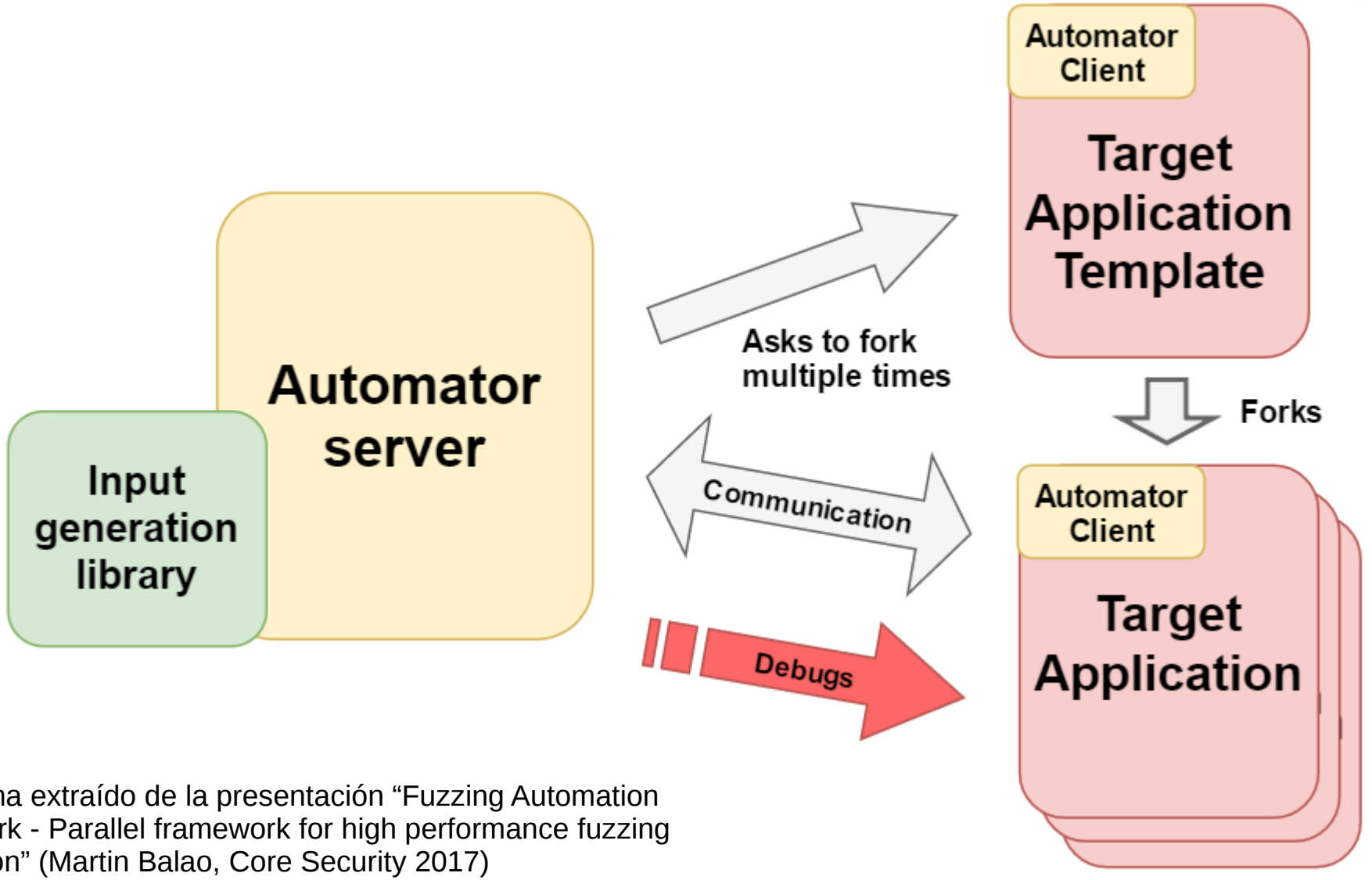




* Diagrama extraído de la presentación “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” (Martin Balao, Core Security 2017)

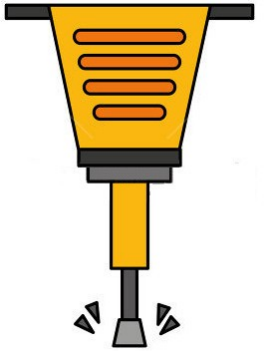
Fuzzing





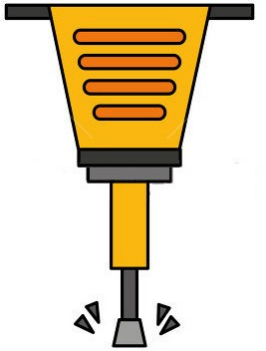
* Diagrama extraído de la presentación “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” (Martin Balao, Core Security 2017)

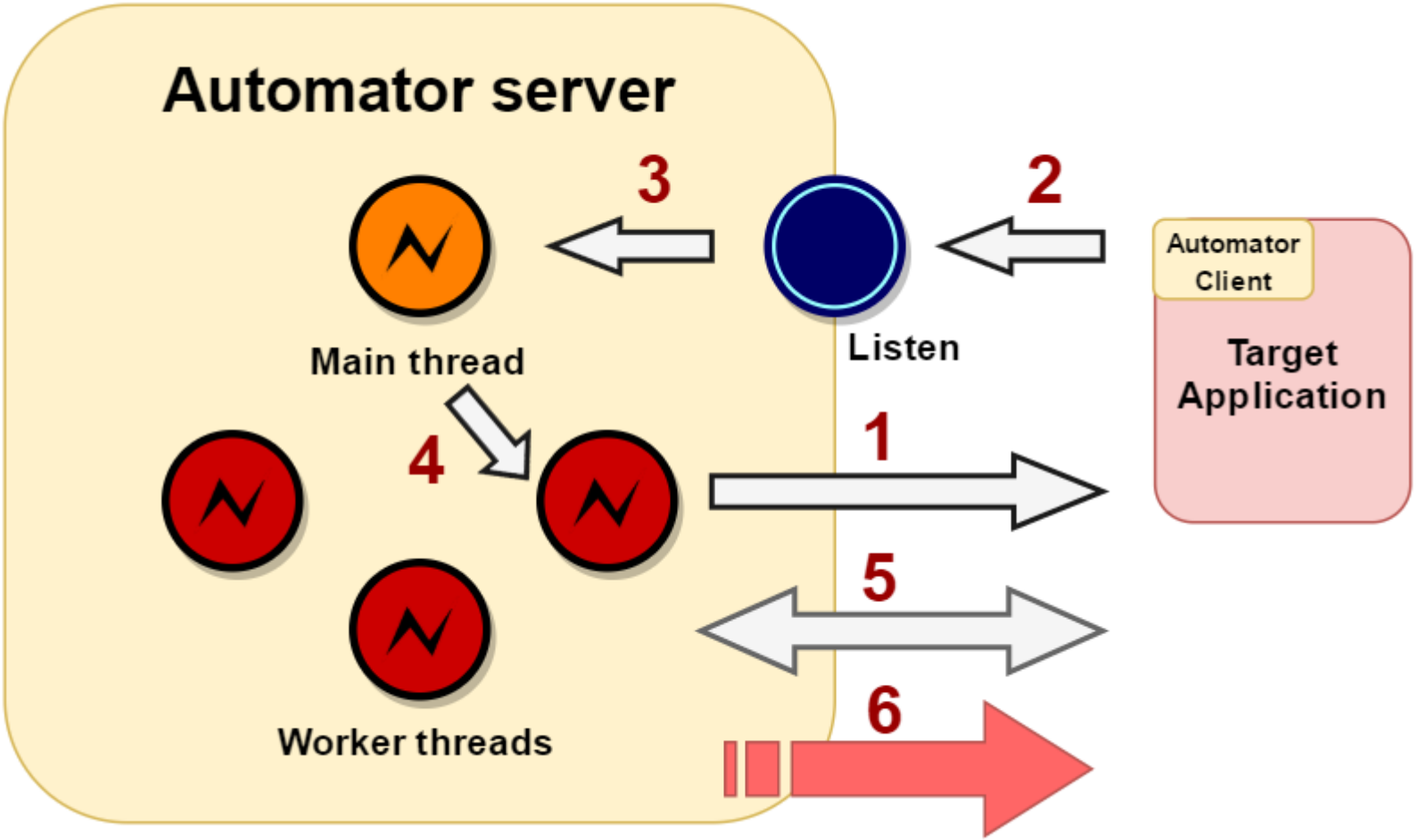
Fuzzing



- 1. Cada Worker Thread spawnea/forkea una Aplicación Targeteada
- 2. La Aplicación Targeteada anuncia su PID
- 3. El Thread Principal handlea el anuncio
- 4. El Thread Principal notifica a un Worker Thread sobre la nueva aplicación
- 5. Una comunicación es establecida entre el Worker Thread y la Aplicación Targeteada
- 6. El Worker Thread debugea a la Aplicación Targeteada

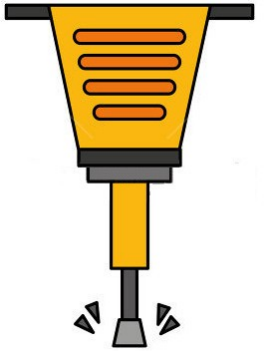
Fuzzing

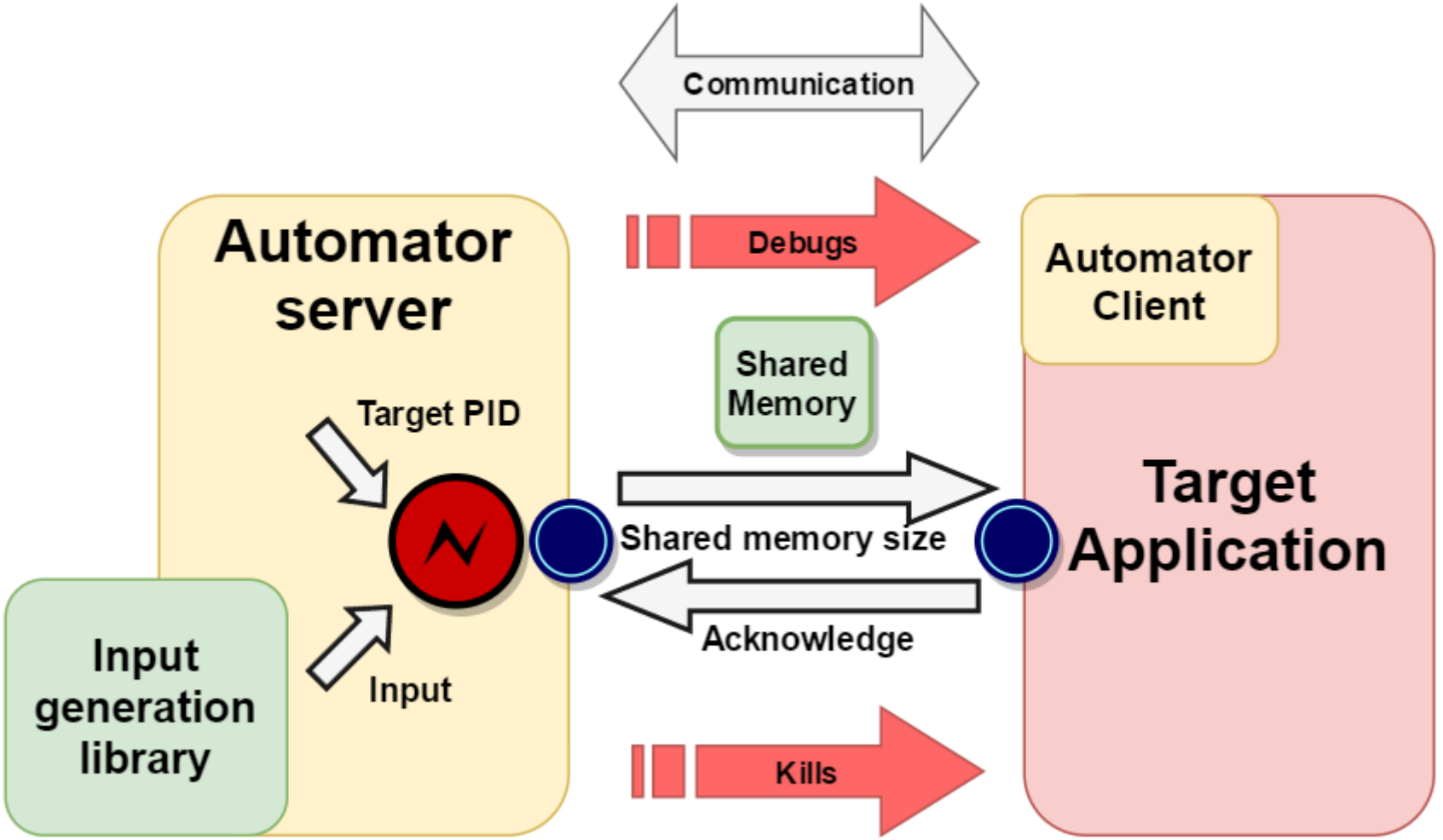




* Diagrama extraído de la presentación “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” (Martin Balao, Core Security 2017)

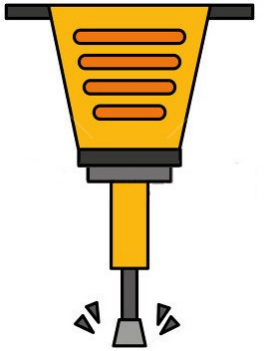
Fuzzing





* Diagrama extraído de la presentación “Fuzzing Automation Framework - Parallel framework for high performance fuzzing automation” (Martin Balao, Core Security 2017)

SMT/SAT Solvers



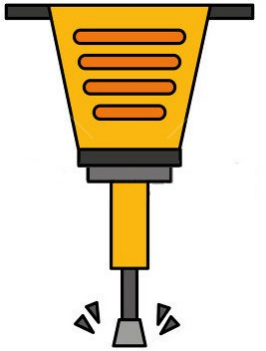
- Resolvers de sistemas de ecuaciones
  - SMT (Satisfiability Modulo Theories) solvers toman los problemas en formas arbitrarias. Variables pueden ser int. Usan SAT solvers como backends

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

  - SAT solvers toman problemas en Forma Normal Conjuntiva (lógica booleana). Operadores booleanos. Variables son true o false

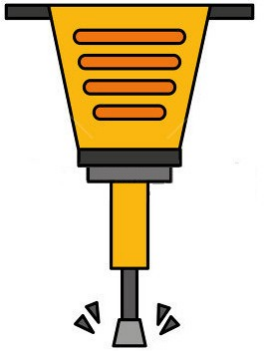
$$\neg A \wedge \left(B \vee C \right)$$

SMT/SAT Solvers



- 3 posibles estados de la solución:
  - No se puede satisfacer
  - Se puede satisfacer (y uno o más ejemplos de solución)
  - ¡No sé! ¿Timeout?
- No es nada nuevo, pero el poder de cómputo hace que ahora se puedan resolver problemas que antes no
- Tiene aplicación a una infinidad de problemas
- z3 es una librería que tiene SMT/SAT solvers. Desarrollada en C++ pero con bindings para múltiples lenguajes (Python, .NET, Java, etc.)

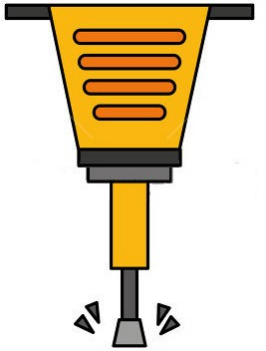
SMT/SAT Solvers



- ¿Cómo resolvemos este sistema de ecuaciones?

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

SMT/SAT Solvers

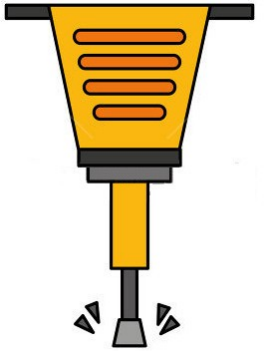


```
#!/usr/bin/python
from z3 import *
x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print s.check()
print s.model()
```

sat

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

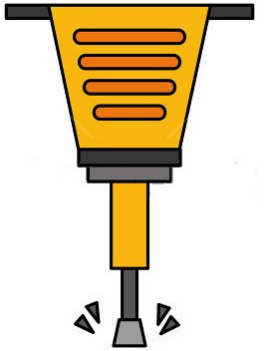
SMT/SAT Solvers



- ¿Cómo resolvemos este Sudoku?

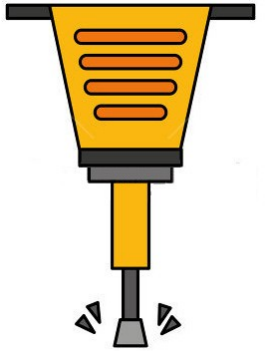
		5	3					
8							2	
	7			1		5		
4					5	3		
	1			7				6
		3	2				8	
	6		5					9
		4					3	
					9	7		

SMT/SAT Solvers



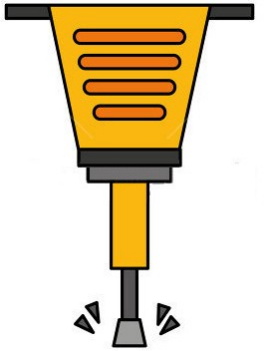
- Hay que llenar los espacios del tablero con números del 1 al 9
- Los números no se puede repetir:
  - Por fila
  - Por columna
  - Por sub-cuadrante
- ¿Podemos plantear este problema de forma tal que sea adecuado para un SMT solver? ¿Cómo modelamos el problema y las restricciones?

SMT/SAT Solvers



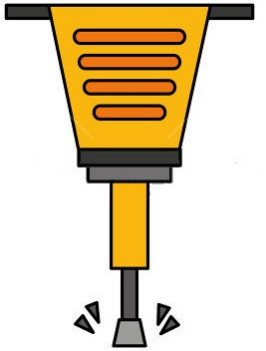
- Modelar el tablero como matriz de Int ([][]): cells=[[Int('cell%d%d' % (r, c)) for c in range(9)] for r in range(9)]
- Agregar restricciones para las celdas que ya tienen un valor asignado: s.add(cells[current_row] [current_column]==int(i))
- Agregar restricciones a cada celda para que la solución este entre 1 y 9: s.add(cells[r][c]>=1), s.add(cells[r][c]<=9)

SMT/SAT Solvers



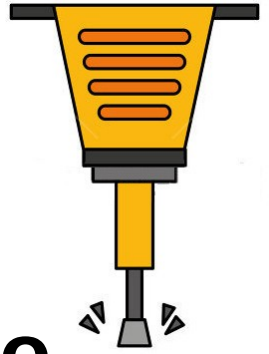
- Agregar restricción para unicidad de columnas y filas: s.add(Distinct(cells[r][0],… cells[r][8])) y s.add(Distinct(cells[0][c],… cells[8][c]))
- Agregar restricción para unicidad por sub-cuadrantes: s.add(Distinct(cells[r+0][c+0]…))
- Chequear si hay solución: s.check()
- Obtener solución: m=s.model()

SMT/SAT Solvers



- ¿Cómo resolvemos este buscaminas?

SMT/SAT Solvers

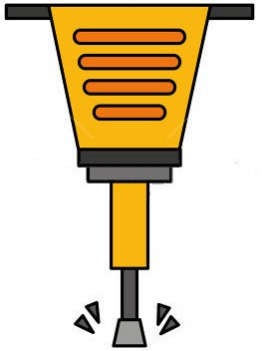


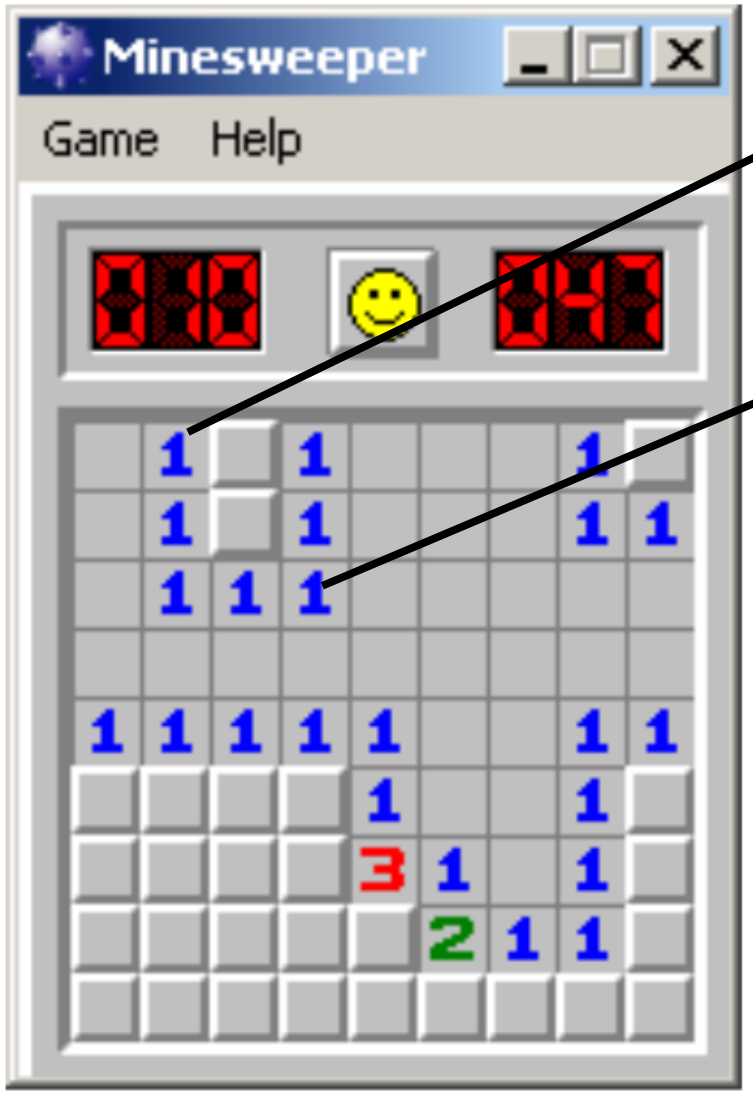
1) ¿Es seguro destapar acá?

2) ¿Y acá?

Supongamos que hay una bomba en cada lugar, ¿podemos satisfacer las restricciones impuestas por las celdas linderas?

SMT/SAT Solvers



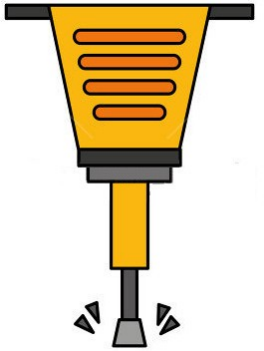


Este 1 pone la siguiente condición: 1) + 2) = 1

Este 1 pone la siguiente condición: 2) = 1

Si suponemos que la bomba está en 1), agregamos la siguiente condición: 1) = 1

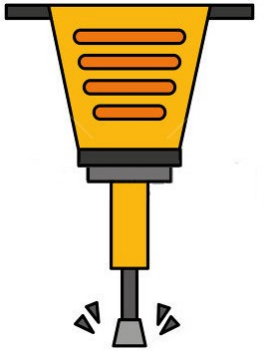
SMT/SAT Solvers



El SMT solver nos dice que el sistema de ecuaciones no tiene solución. Por lo tanto, la bomba no está en 1)

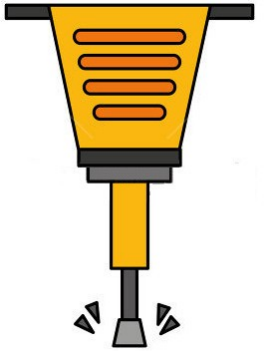
Si hay al menos una solución, **no podemos decidir** si hay o no una bomba

SMT/SAT Solvers



- Es importante modelar adecuadamente el problema y formular la pregunta en términos que el SMT solver pueda responderla (en un tiempo razonable)
- También podemos resolver problemas de optimización en z3

SMT/SAT Solvers

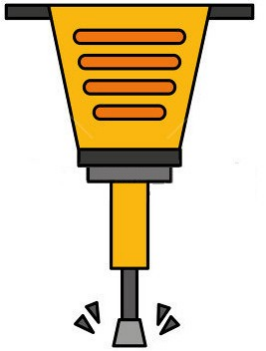


- Crackeando un cipher text (plain text XOR key) con z3

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

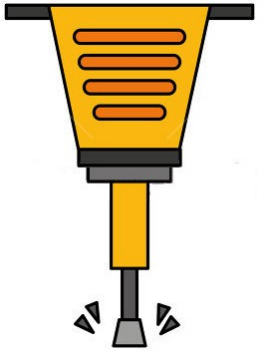
Tabla de verdad de XOR

SMT/SAT Solvers



- Asumamos que el plain text es un texto en inglés. El largo de la clave es desconocido, pero significativamente menor en relación al cipher text
- La estrategia es probar diferentes largos de claves y para cada uno maximizar la cantidad de caracteres alfabéticos
- Necesitamos agregar las restricciones propias de la operación XOR y de la repetición de la clave de forma periódica. Ej. si la clave tiene un largo 5, el byte 0 de la clave se va a XORear con el cipher text en las posiciones múltiplo de 5

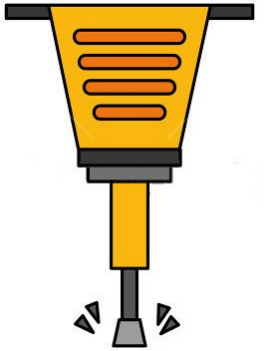
SMT/SAT Solvers



- Variables para modelar el problema

```
# 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 para modelar el problema

Ejemplo (key length = 5)

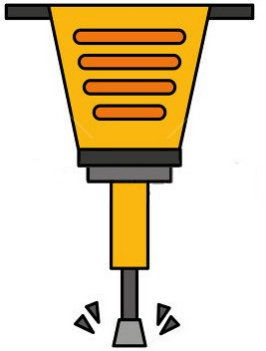
BitVec (8 bits)

- **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

Queremos maximizar el valor de la suma de az_in_plain

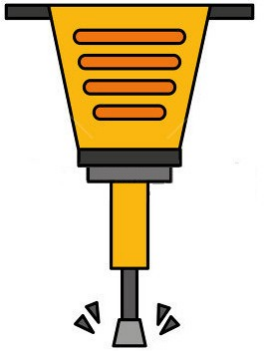
SMT/SAT Solvers



- Restricciones del problema

```
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))
```

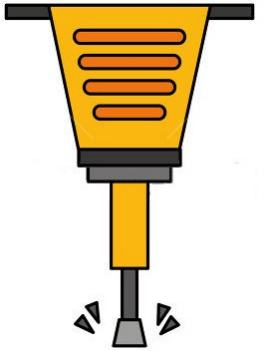
SMT/SAT Solvers



- Solución

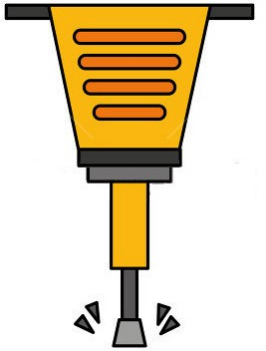
```
s=Optimize()

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



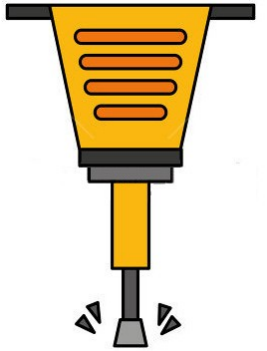
- Solución
  - Se pueden optimizar múltiples variables al mismo tiempo
  - Podemos asumir que la aparición de ciertas letras juntas es más probable y usarlo como vector de optimización
  - Podemos inclusive ponderar los vectores de optimización para asignarles diferente relevancia y “educar” la búsqueda de soluciones

SMT/SAT Solvers



- ¿Cómo funcionan por dentro los SMT/SAT solvers?
  - Teorías comunes
    - Bit Vectors
      - Ideales para representar tipos de datos con rangos finitos. Ej. enteros de 32 bits. Esto permite modelar “overflows” y “underflows”
    - Arrays
      - Largo no-fijo
    - Enteros
    - Funciones no-interpretadas
      - Dado los mismos inputs, se devuelve el mismo output

SMT/SAT Solvers

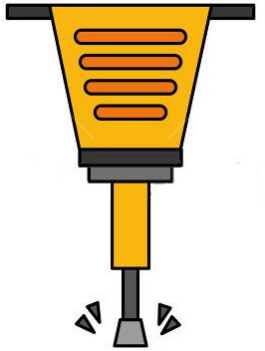


- ¿Cómo funcionan por dentro los SMT/SAT solvers?
  - Base de restricciones en forma normal conjuntiva (todas las fórmulas booleanas pueden reescribirse de esta manera)

$$x_1 \vee x_2 \vee x_3$$

  - El SAT solver asigna un valor de verdad a una variable, y empieza a hacer deducciones basado en eso

SMT/SAT Solvers



- ¿Cómo funcionan por dentro los SMT/SAT solvers?

$$x_1 = true$$

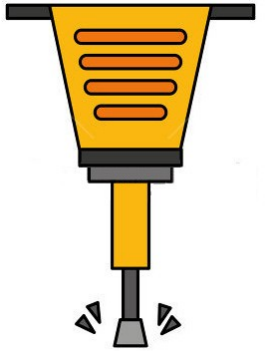
$$-x_1 \vee x_7 \Rightarrow x_7 = true$$

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

...

- Puede asignar valores a todas las variables sin violar restricciones o puede llegar a una contradicción. Si llega a una contradicción, debe resumirla en una cláusula y agregarla a la base de restricciones para no volver a incurrir en ella

SMT/SAT Solvers



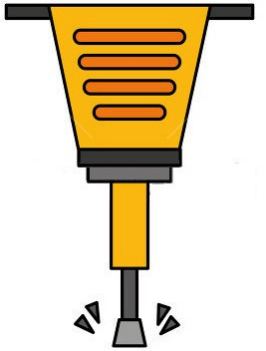
- ¿Cómo funcionan por dentro los SMT/SAT solvers?

$$x>5 \wedge y<5 \wedge (y>x \vee y>2)$$

  - Una parte de esta fórmula requiere razonamiento sobre un dominio específico (ej. conjunto de los enteros) y otra parte es lógica booleana expresable en forma normal conjuntiva (SAT solver)

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

SMT/SAT Solvers



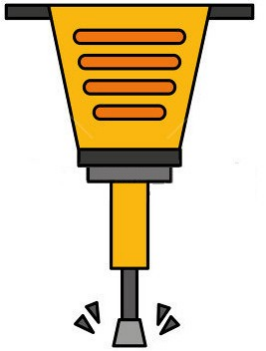
- ¿Cómo funcionan por dentro los SMT/SAT solvers?

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

SAT SOLVER

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

SMT/SAT Solvers



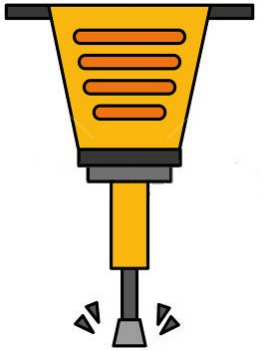
- ¿Cómo funcionan por dentro los SMT/SAT solvers?

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

Theory Solver (linear arithmetic)

NO

SMT/SAT Solvers



- ¿Cómo funcionan por dentro los SMT/SAT solvers?

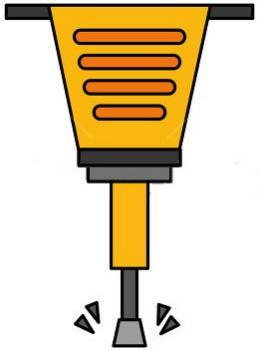
$$F1 \wedge F2 \wedge (F3 \vee F4)$$

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

SAT SOLVER

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

SMT/SAT Solvers



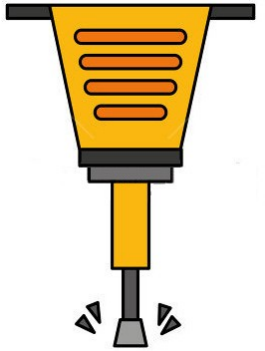
- ¿Cómo funcionan por dentro los SMT/SAT solvers?

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

Theory Solver (linear arithmetic)

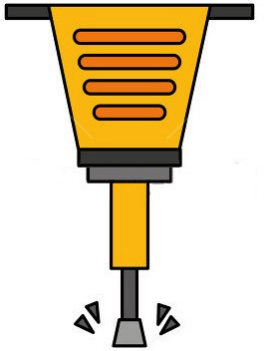
SÍ $x=6,\ y=3$

Ejecución Simbólica



- ¿Cómo pueden ayudar los SMT/SAT solvers a la búsqueda de vulnerabilidades en código?
  - Ejecución simbólica
    - Técnica para el análisis de programas
    - ¿Cómo será el comportamiento en un conjunto potencialmente infinito de entradas?
    - Mejorar la cobertura de código
    - Cuando encuentra un problema, puede proveer un conjunto de inputs para reproducirlo (a diferencia del análisis estático)

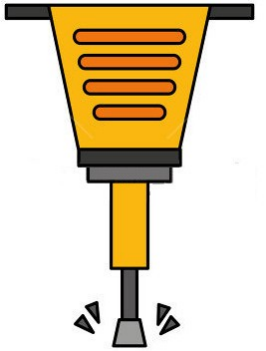
Ejecución Simbólica



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

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

    if (t < x) {
        assert false;
    }

}
```

¿Hay un par de entradas x, y que permitan triggerear el assertion?

Ejecución Simbólica

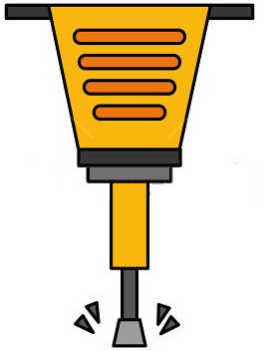


Caracterización del estado del programa: 3 variables de estado

x	y	t
4	4	0
4	4	4

No se triggerea el assertion: x == t

Ejecución Simbólica

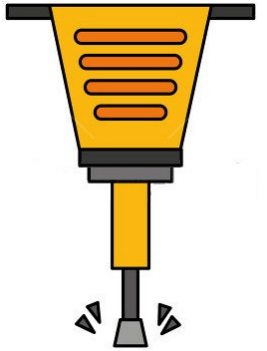


x	y	t
2	1	0
2	1	2

No se triggerea el assertion: x == t

Pero, ¿cómo podemos estar seguros de que no hay inputs para los cuales sí se triggerea el assertion?

Ejecución Simbólica

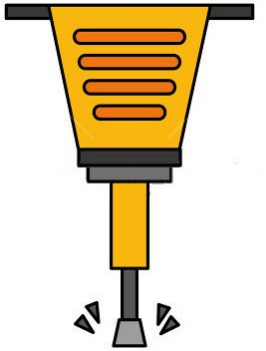


- Redefinimos el estado del programa, mapeando las variables desconocidas (x, y) a valores simbólicos (χ, y)

x	y	t
χ	y	0
χ	y	t_0

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

Ejecución Simbólica



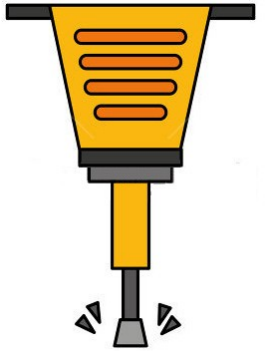
- ¿Es posible satisfacer las siguientes restricciones? ¿Hay una solución para este sistema de ecuaciones?

$$t_0 < x$$

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

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

Ejecución Simbólica



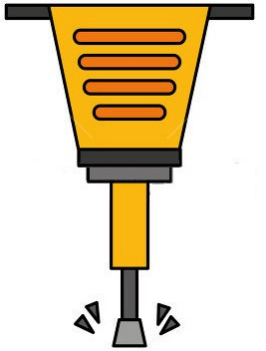
- ¿Es posible satisfacer las siguientes restricciones? ¿Hay una solución para este sistema de ecuaciones?

$$t_0 < x$$

$$(x > y) \Rightarrow t_0 = x$$ ✗

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

Ejecución Simbólica



- ¿Es posible satisfacer las siguientes restricciones? ¿Hay una solución para este sistema de ecuaciones?

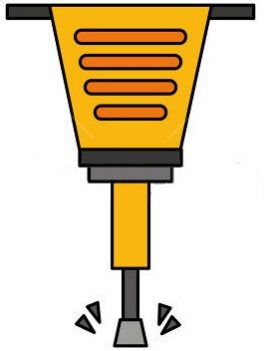
$$t_0 < x$$

$$(x > y) \Rightarrow t_0 = x$$ ✗

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

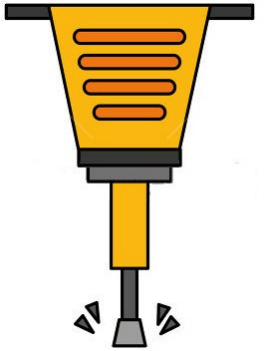
$$(t_0 < t_0)$$ ✗

Ejecución Simbólica



- ¿Es posible satisfacer las siguientes restricciones? ¿Hay una solución para este sistema de ecuaciones?
  - ¡Un SMT/SAT solver nos puede dar esta respuesta!
  - En general, a pesar de que pueden haber muchas variables involucradas en un problema real, no existen tantos grados de libertad: las variables están condicionadas unas por otras
    - Depende del tamaño de la unidad que estamos analizando
    - Si es una función sencilla, podemos analizar todos los caminos de una sola vez

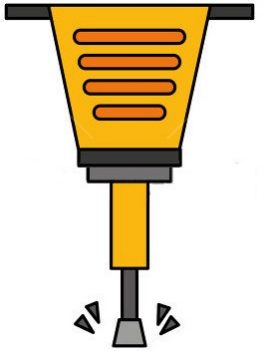
Ejecución Simbólica

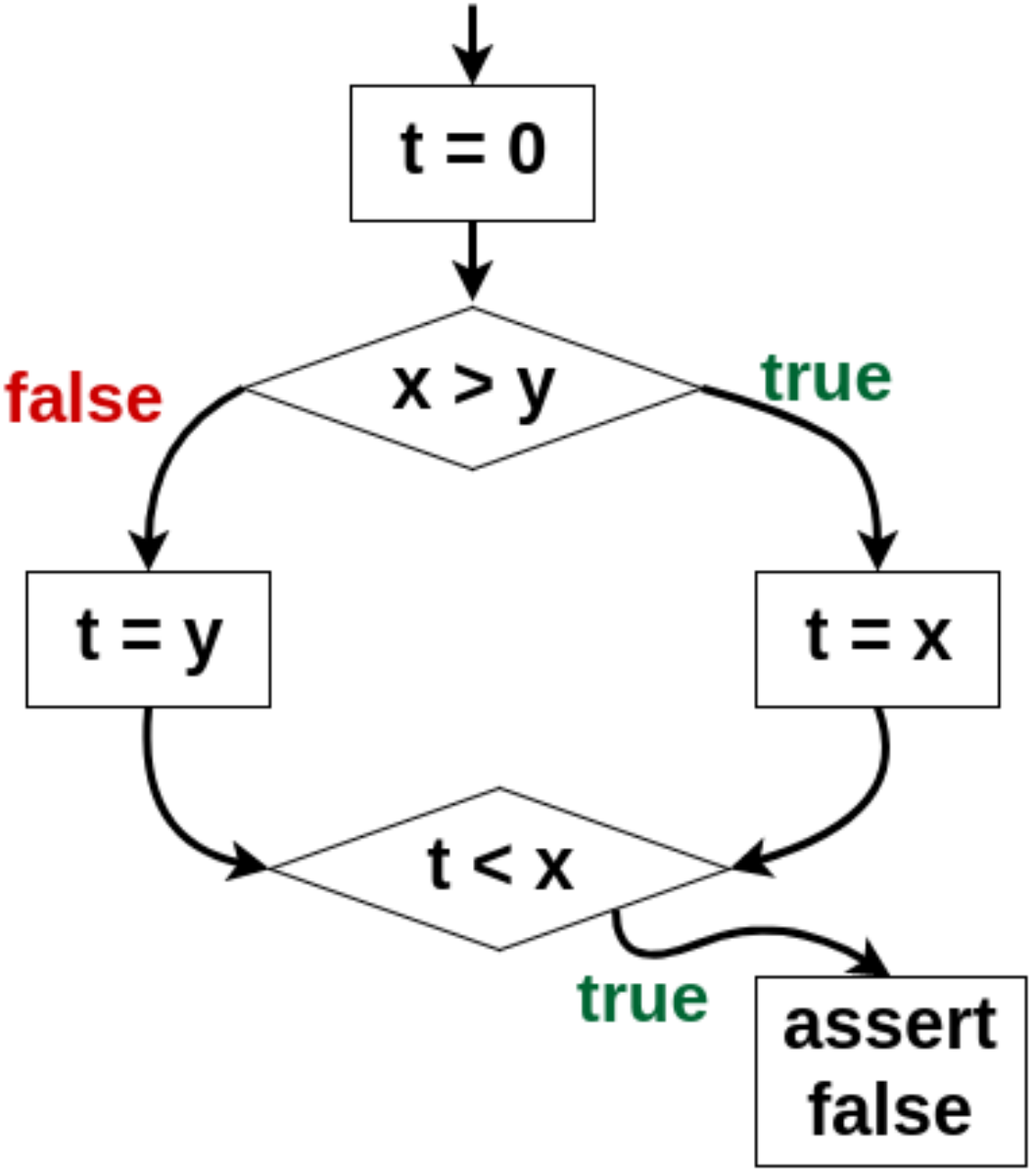


```
#!/usr/bin/python
from z3 import *

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

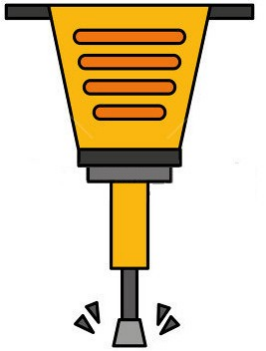
Eiecución Simbólica

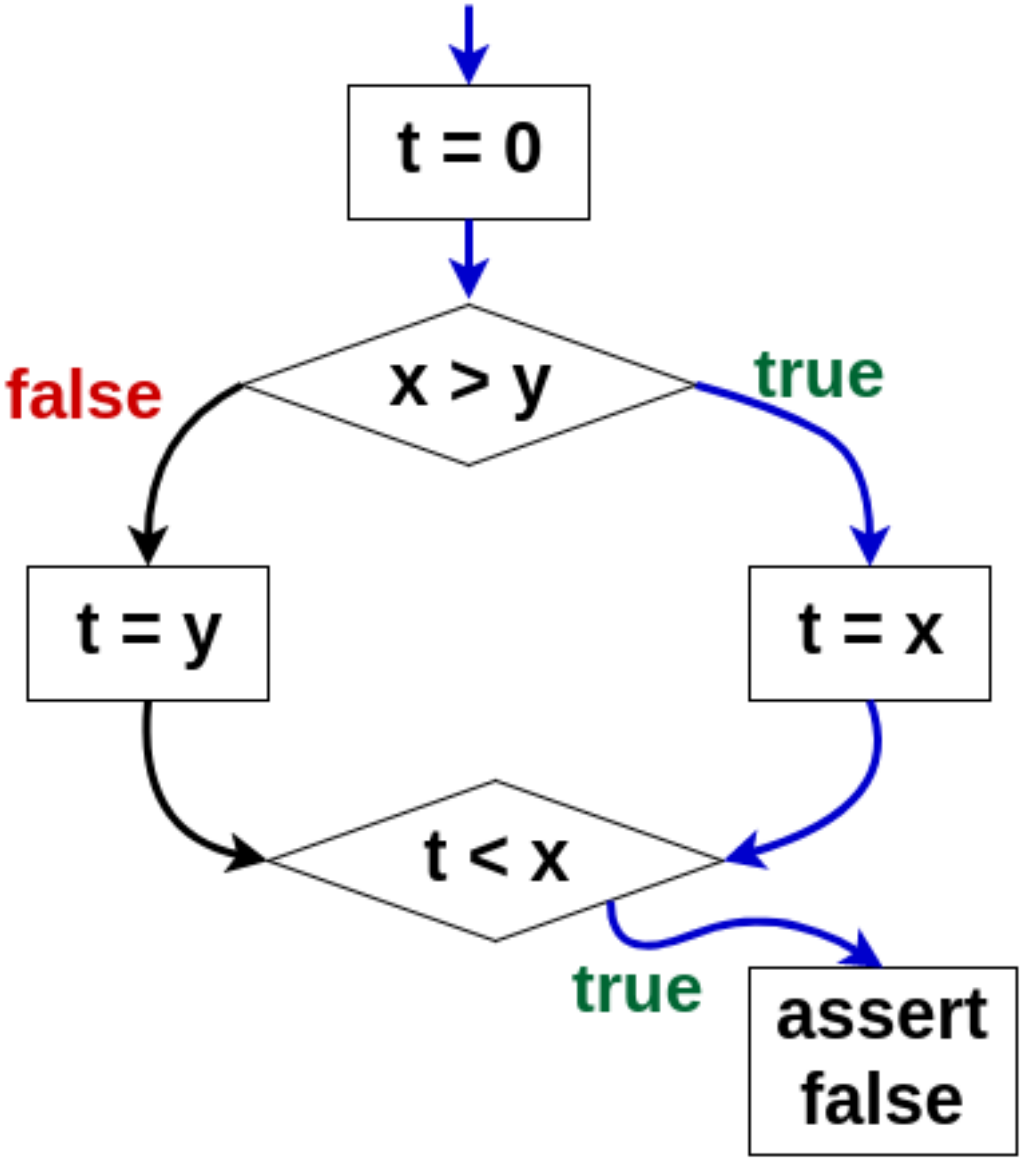




Si el software para analizar es demasiado complejo, podemos hacer exploración de caminos

Ejecución Simbólica





Restricciones:

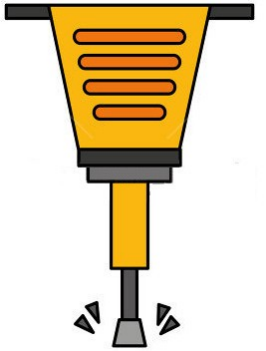
$$t_0 = x$$

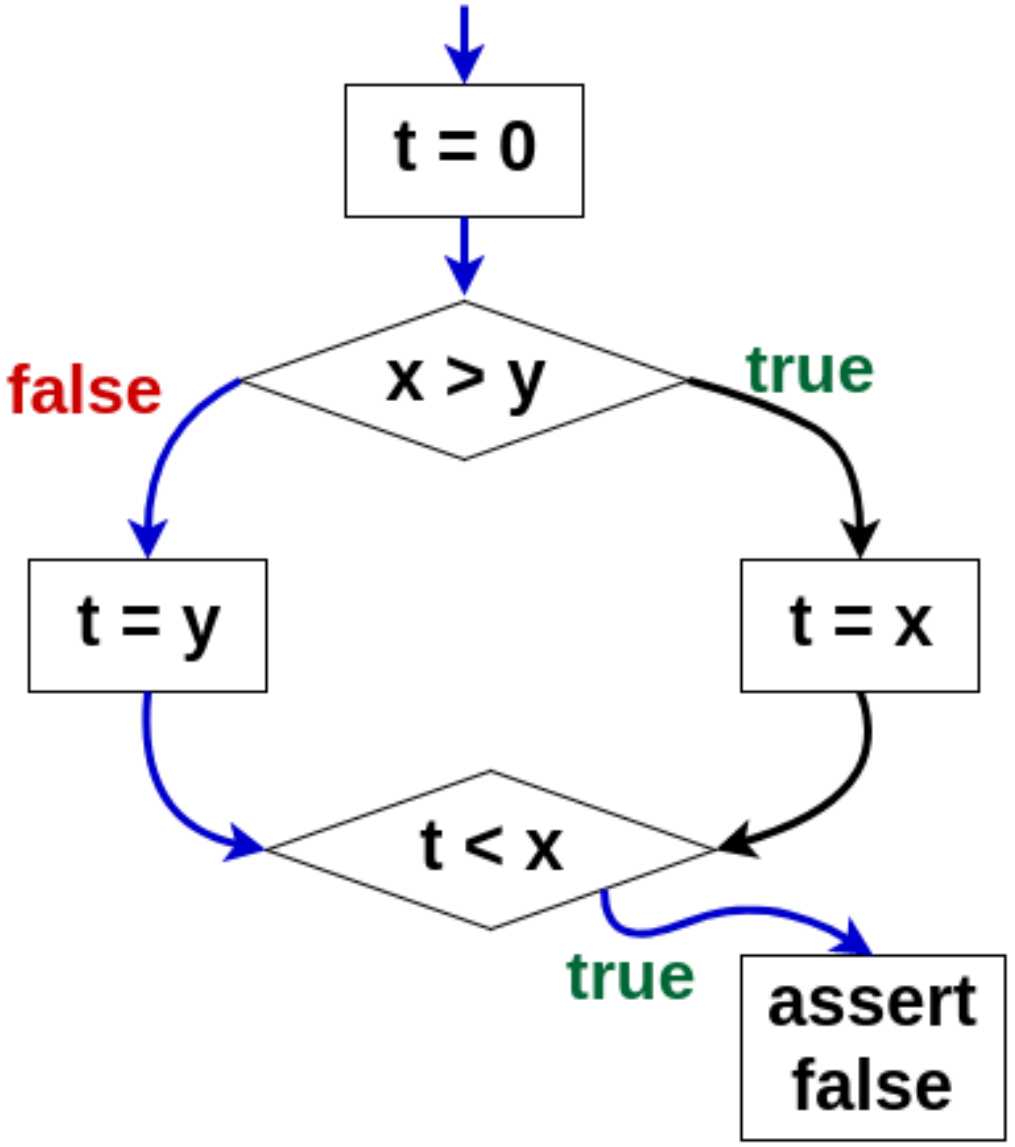
$$t_0 < x$$

Sistema de ecuaciones más simple al explorar solo 1 camino

Nos preguntamos solamente si este camino es factible

Ejecución Simbólica





Restricciones:

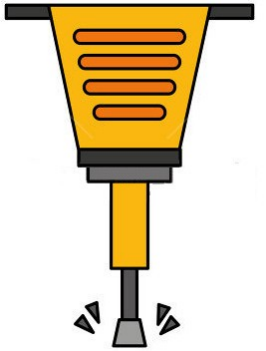
$$x \leq y = t_0$$

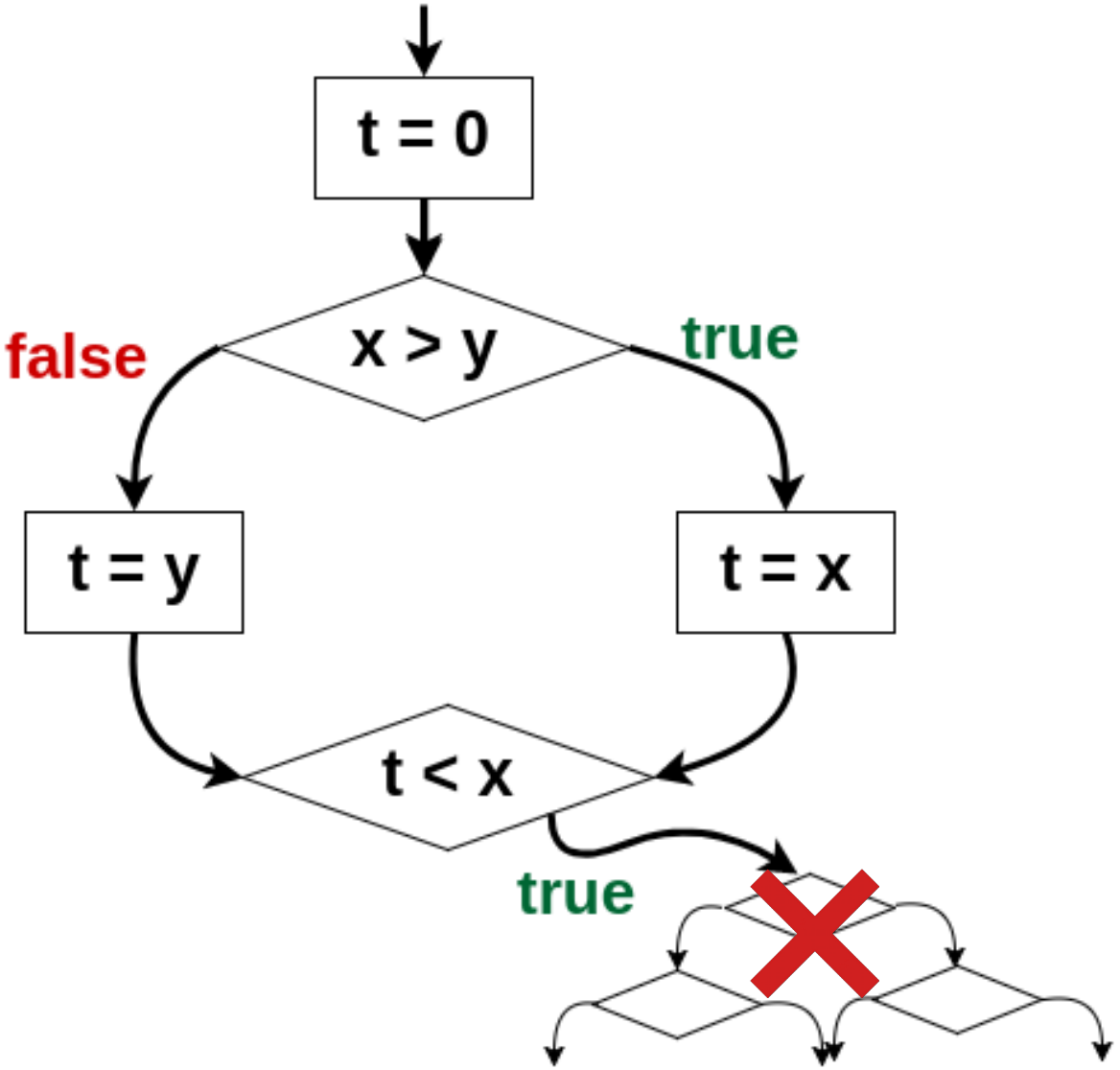
$$t_0 < x$$

Sistema de ecuaciones más simple al explorar solo 1 camino

Nos preguntamos solamente si este camino es factible

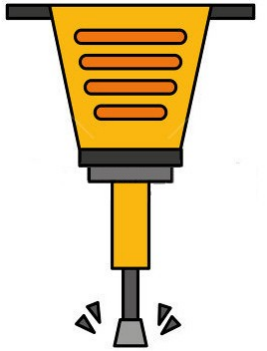
Ejecución Simbólica





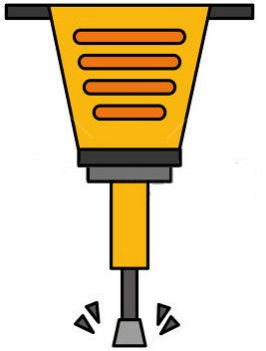
Exploramos más caminos pero cada camino es más simple. Podemos usar estrategias para prunear caminos imposibles.

Ejecución Simbólica



- La ejecución simbólica puede utilizarse como complemento de la ejecución real (fuzzing / testing). Ej:
  - Descubrimos con una herramienta de code-coverage que nos quedó un flujo del programa sin ejercitar haciendo fuzzing
  - Tomamos un caso cercano (generado con input real) y le aplicamos ejecución simbólica a partir de cierto estado para llegar al flujo sin testear

Lab



Lab 6.1: Implementar la función “generate_input” en fuzzer.py para crashear main, sin aplicar ingeniería inversa sobre el binario

- En caso de no lograr crashearlo, aplicar ingeniería inversa para guiar la generación automática de inputs
- En caso de no lograr crashearlo, analizar el código fuente para guiar la generación automática de inputs

Referencias

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