Ingeniería del Software 2

Fuzzing

Juan P. Galeotti

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

ICC Instituto de Ciencias de la Computación

Fuzz Testing

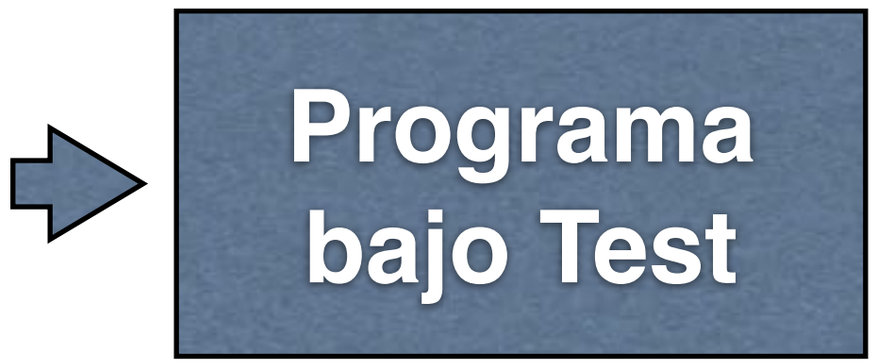
- **Idea**: Estudiar cómo el programa soporta "ruido" en el input
- Fuzzers: Herramientas inputs de un programa con el objeto de:
  - Crashear el programa
  - Encontrar fallas de seguridad
- Inspirado en la vida real

Fuzzing

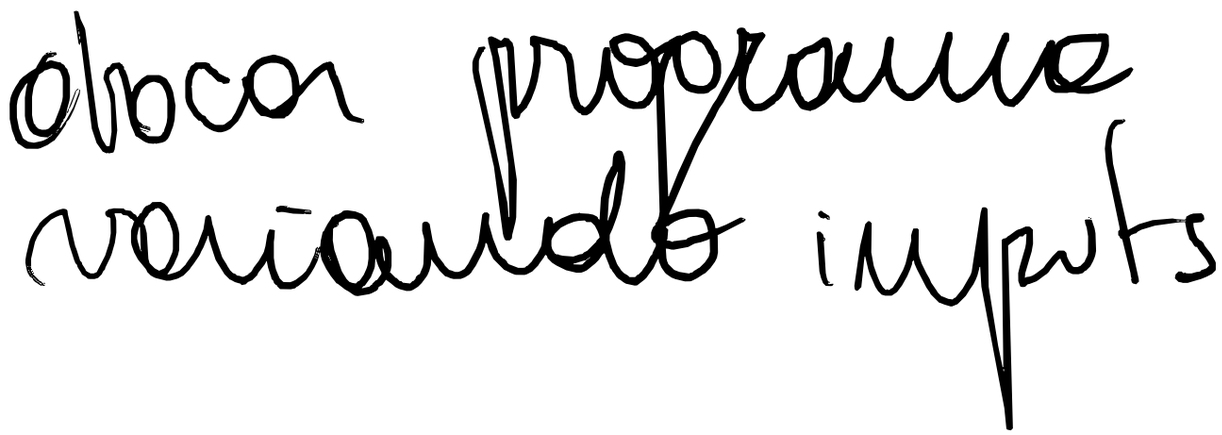
Barton P. Miller

"... in the Fall of 1988, there was a wild midwest thunderstorm ... With the heavy rain, there was noise on the (dial–up) line and that noise was interfering with my ability to type sensible commands to the shell"

Fuzz Testing



“ab’d&gfdfggg”

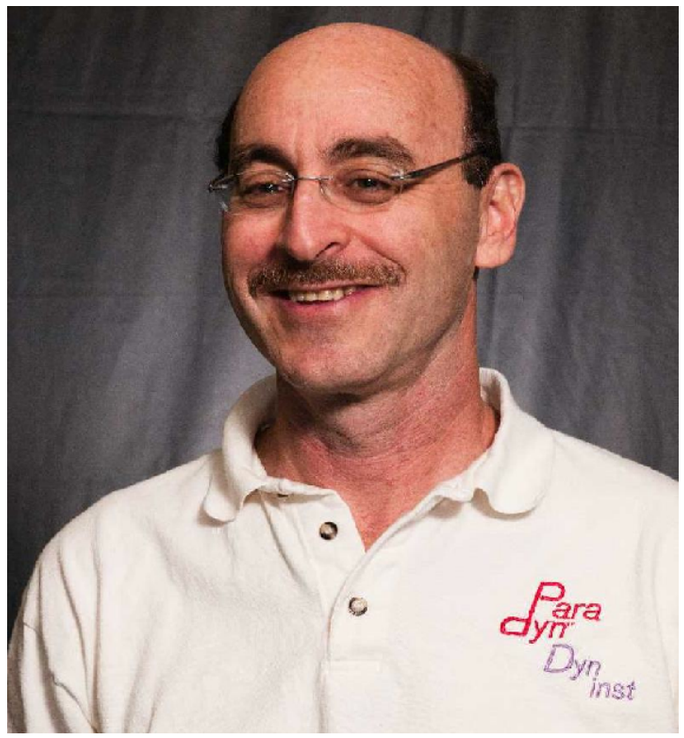


Paper de 1989

An Empirical Study of the Reliability

of

UNIX Utilities



Barton P. Miller
bart@cs.wisc.edu

Lars Fredriksen
L.Fredriksen@att.com

Bryan So
so@cs.wisc.edu

Summary

Operating system facilities, such as the kernel and utility programs, are typically assumed to be reliable. In our recent experiments, we have been able to crash 25-33% of the utility programs on any version of UNIX that was tested. This report describes these tests and an analysis of the program bugs that caused the crashes.

Fuzzing

“ab’d&gfdfggg” grep • sh • sed … 25%–33%

Fuzzing en sus orígenes = Random Testing a nivel de Sistema (System Level)

Causas de Crashes

- Uso Punteros y accesos a Arreglos
- Ausencia de chequeo de códigos de retorno (return codes)
- Y más…

Punteros y Arreglos

¿Qué problema puede existir en este programa?

```
while ((cc = getch()) != c)
{
    string[j++] = cc;
    ...
}
```

No se chequea la longitud máxima del string

Return Codes

```
char rdc()
{
    char lastc;

    do {
        lastc = getchar();
    } while (lastc != ' ' &&
             lastc != '\t');

    return (lastc);
}
```

Si getchar() alcanza EOF

¿Qué ocurre?

Y más…

- Comando "!o%8f" al programa VAX csh
- “o%8f: Event not found.”
- printf(“o%8f: Event not found.”)

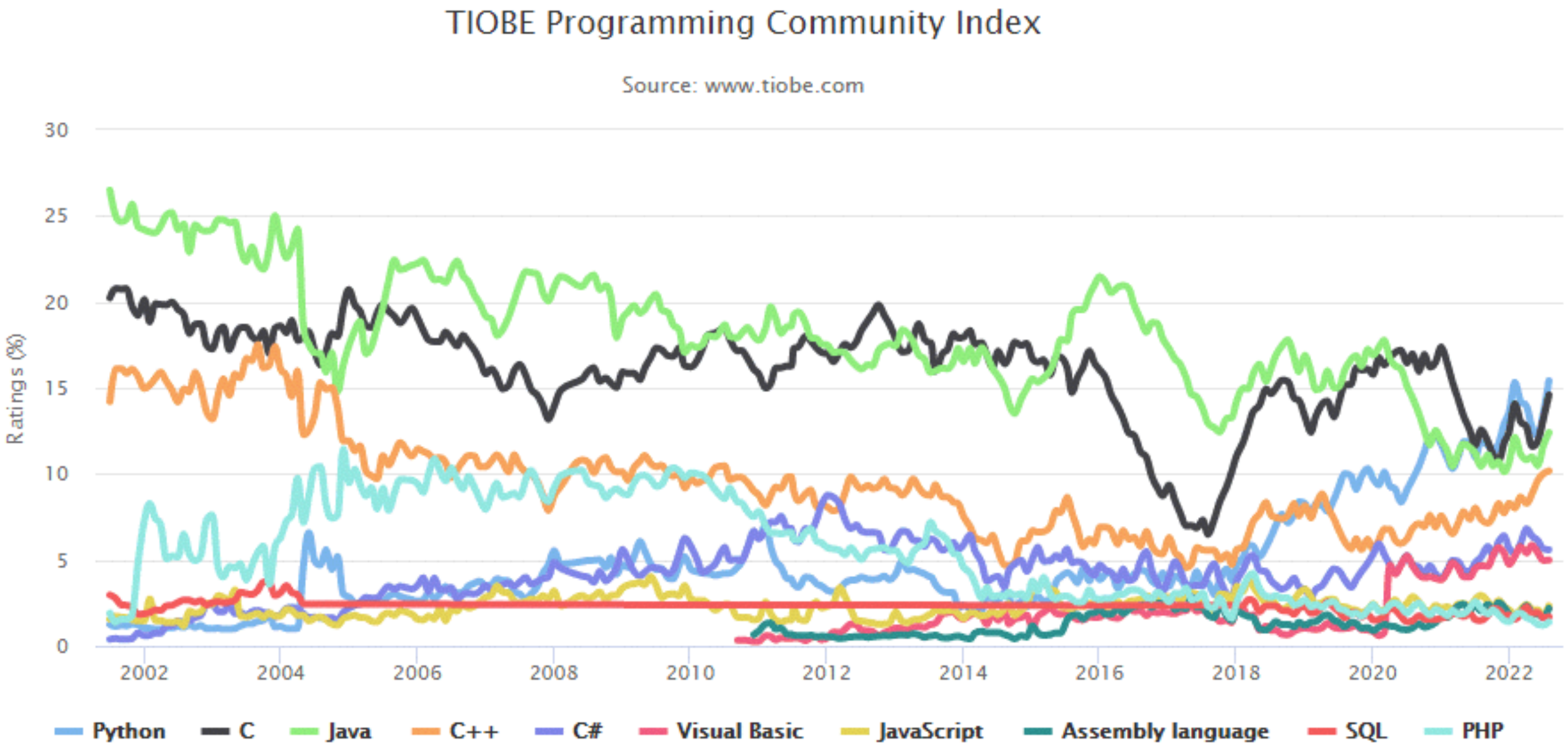
CRASH!

Reglas de la Programación Segura

- Chequear que todos los accesos a arreglos usan índices válidos
- Aplicar bounds en todos los inputs
- Chequear todos los return codes
- Nunca confiar en inputs de 3rd-parties

…pero todo esto generalmente “gratis” con los lenguajes “modernos”

Lenguajes de Programación



El Lenguaje C

Random Black-Box Fuzzing

- Testing Black-box trata al programa bajo test como una "Caja Negra" (No se conoce nada interno del programa)
- Fuzzing, en su inicios, fue concebido como una técnica black-box
  - Random Testing a nivel de Sistema (System Level)
- Fundamentalmente, ejecutar programas con inputs random y ver si podemos "romper cosas"

Random Black-Box Fuzzing

- ¿Qué pasa si aplicamos random black-box fuzzing a este programa?

```
def http_program(url: str) -> bool:
    supported_schemes = ["http", "https"]
    result = urlparse(url)
    if result.scheme not in supported_schemes:
        raise ValueError("Scheme must be one of " +
                         repr(supported_schemes))
    if result.netloc == '':
        raise ValueError("Host must be non-empty")

    # Do something with the URL
    return True
```

Límites del random fuzzing

- Haciendo random fuzzing en general, hay bajas chances de generar inputs válidos

- ¿Qué podemos hacer para aumentar las chances de generar inputs válidos?

Fuzzing

- Idea: Testing a nivel de sistema (i.e. no controlamos el input)
- Buscamos aserciones violentadas, buffers overflows (vulnerabilidad), memory leaks (robustez)
- Casi siempre partimos de una semilla (i.e. conjunto inicial de inputs que serán “fuzzeados”)

Mutation-based fuzzing

agrupar ruido a los inputs

FUZZ SEED

- Partimos de un conjunto de inputs (al que llamaremos seed o semilla)
  - Se elige cada seed y se le aplican una cantidad aleatoria de mutaciones entre 0 y K.
  - Algunas mutaciones posibles:
    - Insertar un caracter
    - Eliminar un caracter
    - "Flipear" un caracter

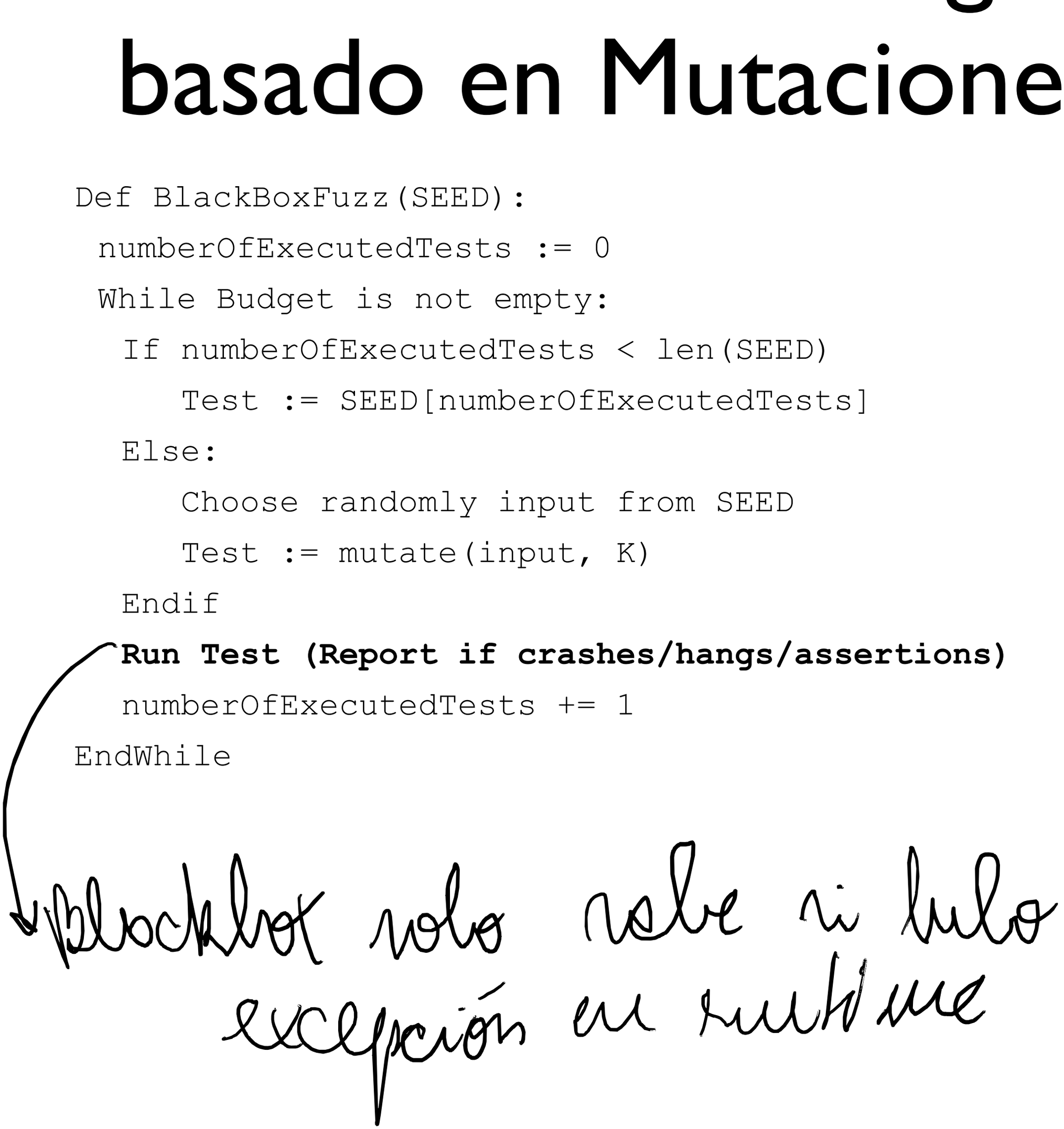
inputs válidos (seed) → se mutan (ruido)

cómo se mejora? → aquellos que aumentan cobertura, seguir explorando

→ los agrego a la seed

Blackbox Fuzzing basado en Mutaciones

```
Def BlackBoxFuzz(SEED):
 numberOfExecutedTests := 0
 While Budget is not empty:
  If numberOfExecutedTests < len(SEED)
     Test := SEED[numberOfExecutedTests]
  Else:
     Choose randomly input from SEED
     Test := mutate(input, K)
  Endif
  Run Test (Report if crashes/hangs/assertions)
  numberOfExecutedTests += 1
EndWhile
```



Mutation-based Fuzzing

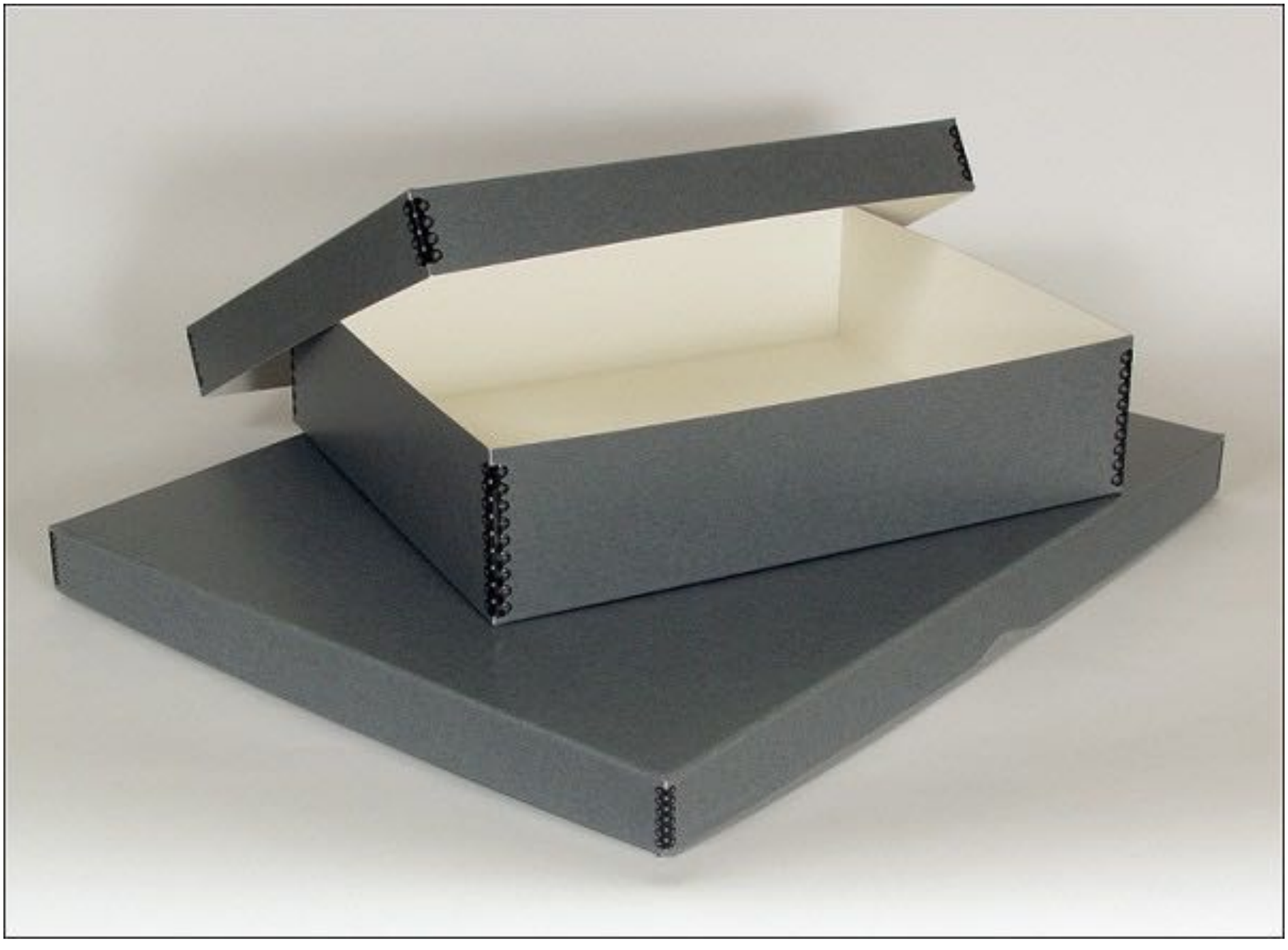
- ¿Qué pasa si aplicamos random black-box fuzzing o mutation-based fuzzing este programa?

```
def crashme(s: str) -> None:
    if len(s) > 0 and s[0] == 'b':
        if len(s) > 1 and s[1] == 'a':
            if len(s) > 2 and s[2] == 'd':
                if len(s) > 3 and s[3] == '!':
                    raise Exception()
```

- ¿Cómo podemos aumentar las chances de construir inputs para alcanzar la excepción usando fuzzing?

↳ preparando inputs a lo SEED
∴ crece espacio de tentativo

Greybox Fuzzing



Greybox Fuzzing

- Idea: Si un input aportó cobertura, lo agrego al corpus de inputs para futuras mutaciones
- Para eso necesitamos medir la cobertura (ej: branches cubiertos, basic blocks cubiertos) del programa bajo test
- Energía: cada elemento del seed posee una "energía" (i.e. probabilidad de elegirlo)

Greybox Fuzzing

```
Def GreyBoxFuzz(SEED):
  numberOfExecutedTests := 0
  init_len := len(SEED)
  While Budget is not empty:
    If numberOfExecutedTests < init_len
       Input := SEED[numberOfExecutedTests]
       Execute Input (Report if crashes/hangs/etc)
  Else:
     Choose randomly input from SEED
     NewInput := mutate(input, K)
     Execute NewInput (Report if crashes/hangs/etc)
     If NewInput adds new Coverage:
       SEED += NewInput
     Endif
  Endif
 numberOfExecutedTests+=1
EndWhile
```

add input to seed (reward good input)

ENERGIA = probabilidad de elegir un input durante campaña de fuzzing.

↳ quiero ⊕ energía p/ input ⊕ rara, ⊖ p/ ⊕ común

Paper de 2016

Coverage-based Greybox Fuzzing as Markov Chain

Marcel Böhme Van-Thuan Pham Abhik Roychoudhury

School of Computing, National University of Singapore, Singapore
{marcel,thuanpv,abhik}@comp.nus.edu.sg

ABSTRACT

Coverage-based Greybox Fuzzing (CGF) is a random testing approach that requires no program analysis. A new test is generated by slightly mutating a seed input. If the test exercises a new and interesting path, it is added to the set of seeds; otherwise, it is discarded. We observe that most tests exercise the same few "high-frequency" paths and develop strategies to explore significantly more paths with the same number of tests by gravitating towards low-frequency paths.

We explain the challenges and opportunities of CGF using a Markov chain model which specifies the probability that fuzzing the seed that exercises path i generates an input that exercises path j. Each state (i.e., seed) has an *energy* that specifies the number of inputs to be generated from that seed. We show that CGF is considerably more efficient if energy is *inversely proportional to the density of the stationary distribution* and *increases monotonically* every time that seed is chosen. Energy is controlled with a power schedule.

We implemented the exponential schedule by extending AFL. In 24 hours, AFLFast exposes 3 previously unreported CVEs that are not exposed by AFL and exposes 6 previously unreported CVEs 7x faster than AFL. AFLFast produces at least an order of magnitude more unique crashes than AFL.

CCS Concepts:

•Security and privacy→Vulnerability scanners; •Software and its engineering→Software testing and debugging;

It turns out that even the most effective technique is less efficient than blackbox fuzzing if the time spent generating a test case takes relatively too long [3]. Symbolic execution is very effective because each new test exercises a different path in the program. However, this effectiveness comes at the cost of spending *significant time doing program analysis and constraint solving*. Blackbox fuzzing, on the other hand, does not require any program analysis and generates several orders of magnitude more tests in the same time.

Coverage-based Greybox Fuzzing (CGF) is an attempt to make fuzzing more effective at path exploration *without* sacrificing time for program analysis. CGF uses lightweight (binary) instrumentation to determine a unique identifier for the path that is exercised by an input. New tests are generated by slightly mutating the provided seed inputs (we also call the new tests as *fuzz*). If some fuzz exercises a new and interesting path, the fuzzer retains that input; otherwise, it discards that input. The provided and retained seeds are fuzzed in a continuous loop, contributing even more seeds.

Compared to symbolic execution, CGF does not require program analysis which brings several benefits. There is *no imprecision*, for instance, in the lifting of the control-flow graph from the program binary or the encoding of the path condition as SMT formula. CGF is more *scalable* because the time to generate a test does not increase with the program size. CGF is highly *parallelizable* because the retained

"Boosted" Greybox fuzzing

- Observación: La mayoría de los inputs durante greybox fuzzing el mismo camino de ejecución
- Idea: Aumentar la probabilidad de elegir un input de la semilla de acuerdo a las chances de descubrir otros caminos en el CFG

Los inputs recorren ciertos caminos,
⇒ queiro ⊕ energía p/ aquellos que recorran caminos raros

"Boosted" Greybox fuzzing

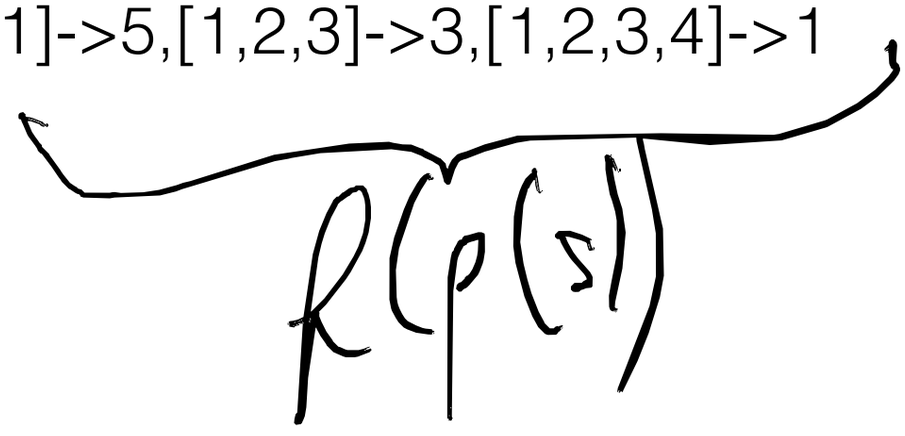
- La energía de un input s se define como e(s) donde p(s) es el camino que recorrió la ejecución de s y f(p(s)) es la frecuencia de apariciones de un camino en el test suite

$$e(s) = \frac{1}{f(p(s))^a}$$

inversamente proporcional a la #veces que veo el camino

```
def crashme(s: str) -> None:
1:     if len(s) > 0 and s[0] == 'b':
2:         if len(s) > 1 and s[1] == 'a':
3:             if len(s) > 2 and s[2] == 'd':
4:                 if len(s) > 3 and s[3] == '!':
5:                     raise Exception()
```

	Input	Camino	Frecuencias
1	aaaa	[1]	[1]->1
2	aaab	[1]	[1]->2
3	aaac	[1]	[1]->3
4	aaad	[1]	[1]->4
5	aaa.	[1]	[1]->5
6	baa.	[1,2,3]	[1]->5,[1,2,3]->1
7	baad	[1,2,3]	[1]->5,[1,2,3]->2
8	bacd	[1,2,3]	[1]->5,[1,2,3]->3
9	badc	[1,2,3,4]	[1]->5,[1,2,3]->3,[1,2,3,4]->1



$$e(s) = \frac{1}{f(p(s))^a}$$

[1]->5,[1,2]->3,[1,2,3]->1

Sea a=3

↳ velocidad de desgaste

	Input	Camino	Energía
1	aaaa	[1]	1:(5^3)
2	aaab	[1]	1:(5^3)
3	aaac	[1]	1:(5^3)
4	aaad	[1]	1:(5^3)
5	aaa.	[1]	1:(5^3)
6	baa.	[1,2]	1:(3^3)
7	baad	[1,2]	1:(3^3)
8	bacd	[1,2]	1:(3^3)
9	badc	[1,2,3]	1:(1^3)

$$e(s) = \frac{1}{f(p(s))^a}$$

[1]->5,[1,2]->3,[1,2,3]->1

Sea a=3

	Input	Camino	Energía
1	aaaa	[1]	1:(5^3)
2	aaab	[1]	1:(5^3)
3	aaac	[1]	1:(5^3)
4	aaad	[1]	1:(5^3)
5	aaa.	[1]	1:(5^3)
6	baa.	[1,2]	1:(3^3)
7	baad	[1,2]	1:(3^3)
8	bacd	[1,2]	1:(3^3)
9	badc	[1,2,3]	1:(1^3)

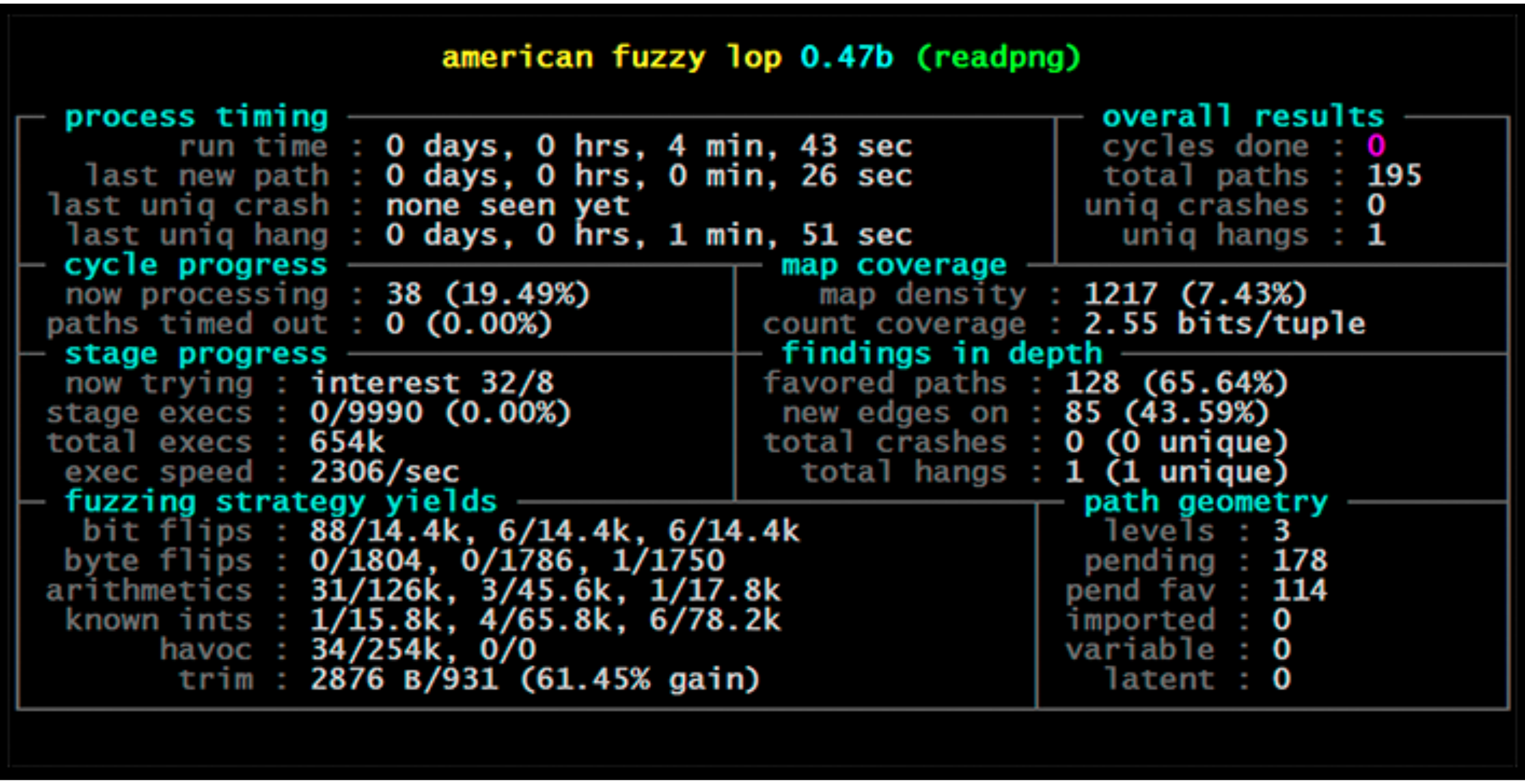
Total Energía = 5 x 1:(5^3) + 3 x 1:(3^3) + 1 x 1:(1^3)

Probabilidad de seleccionar s = e(s) / Total Energía

Boosted Greybox Fuzzing

```
Def BoostedGreyBoxFuzz(SEED):
  energy:= {}
  init_len = len(SEED)
  numberOfExecutedTests := 0
  While Budget is not empty:
    If numberOfExecutedTests < init_len
       Input := SEED[numberOfExecutedTests]
       Execute Input (Report if crashes/hangs/etc)
   Else:
      Choose input from SEED using energy(…)
      newInput := mutate(input, K)
      Execute NewInput (Report if crashes/hangs/etc)
      If NewInput adds new Coverage:
        SEED += NewInput
      Endif
   Endif
  numberOfExecutedTests += 1
  Update energy(s) for each s in SEED
Endwhile
```

American Fuzzy Lop



American Fuzzy Lop (AFL)

≠ greybox

- AFL es uno de los dos fuzzers más populares
- AFL aplica una variante del algoritmo de greybox fuzzing
- AFL funciona sobre programas C/C++ que pueden ser traducidos a LLVM (para medir cobertura de bloques básicos)
- A diferencia del algoritmo de greybox fuzzing que vimos antes, AFL es determinístico.

↳ not random

Fuzzing con AFL

1) Encola **inputs iniciales** provistos por el usuario a la cola Q,

2) Tomar un input de la cola Q

3) Intentar **reducir** el input a su menor tamaño si alterar su capacidad de cobertura,

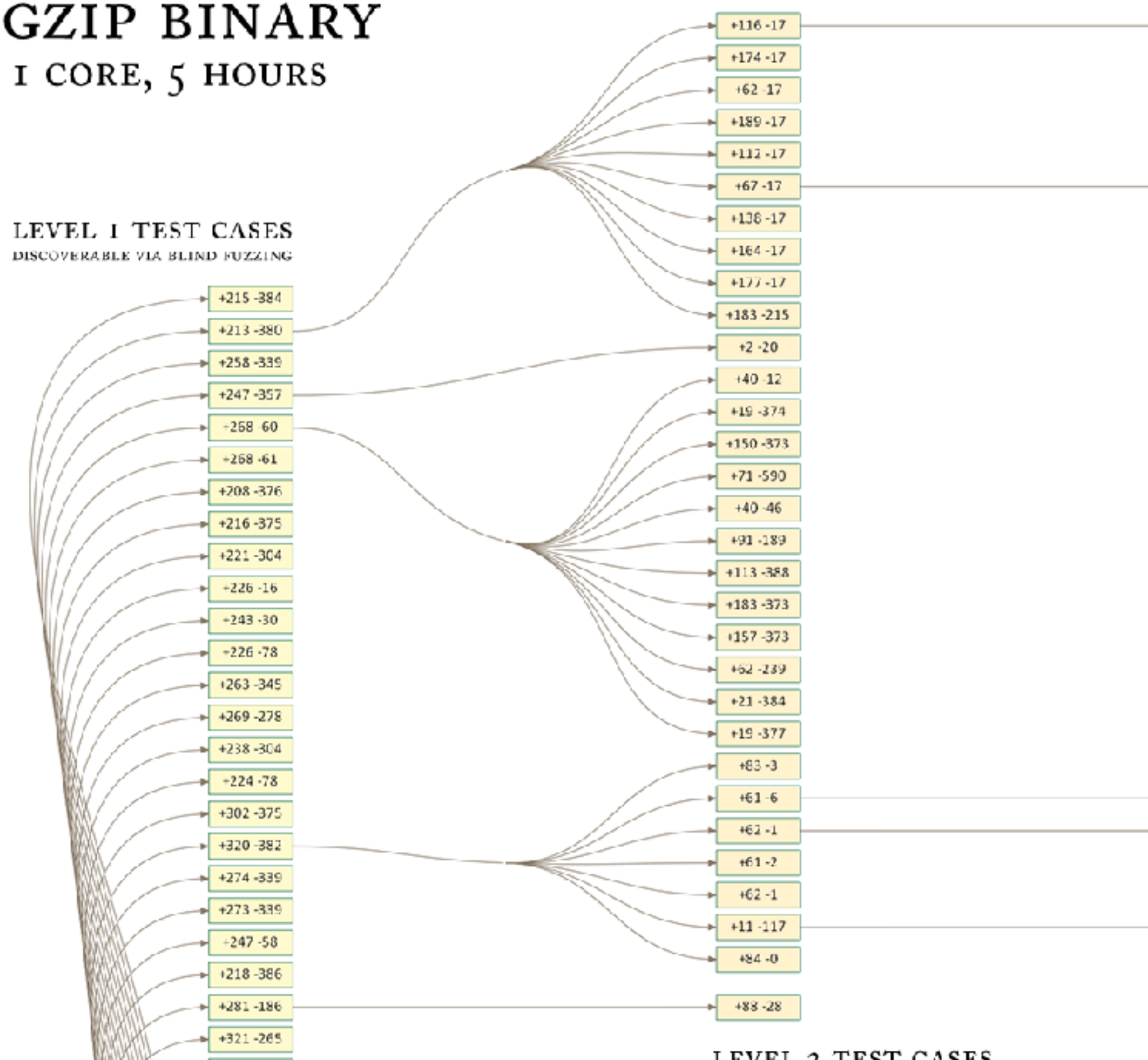
4) Aplicar un conjunto de mutaciones (cambios) al input utilizando una variedad de **estrategias tradicionales de fuzzing**,

5) Si alguno de los nuevos inputs resulta en un aumento de cobertura, agregar el input a la cola Q.

6) Volver a 2).

AFL-FUZZ, GZIP BINARY

2,000 EXECS/SEC, 1 CORE, 5 HOURS



AFL-FUZZ, GZIP BINARY

2,000 EXECS/SEC, 1 CORE, 5 HOURS

Initial test case

LEVEL 1 TEST CASES

DISCOVERABLE VIA BLIND FUZZING

+215 -384
+213 -380
+258 -339
+247 -357
-268 60
+268 -61
+208 -376
+216 -375
+221 -304
-226 -16
+248 -30
+226 -78
+263 -345
+269 -278
+238 -304
-224 -78
+302 -375
+320 -382
+274 -339
+273 -339
-247 -58
+218 -386
+281 -186
+321 -265
+263 -277
-230 -43
+228 -393
+741 -60
+273 -185
-242 -60
+260 -341
-239 -60
+322 -382
+340 -154
+228 -316
+245 -78
-275 -42
-248 -40
-248 -42
-229 -63
+262 -306
-226 -80
+222 -386
-281 -60
+244 -379
+297 -205
+244 -304
+280 -382
+227 -363
+259 -341
+306 -375
+252 -382
+294 -205
+228 -361
-238 -16
+246 -285
+232 -357
+325 -382
+224 -5
+304 -205
+227 -371
+283 -339
+228 -374
+227 -368
+306 -375
+65 -372
+219 -5
+316 -158
+326 -382
+318 -382
+239 -304
+319 -196

LEVEL 2 TEST CASES

DERIVED BY SEEDING THE FUZZER WITH TEST CASES ISOLATED ON PREVIOUS LEVEL

+116 -17
+174 -17
+62 -17
+189 -17
+112 -17
+67 -17
+138 -17
+164 -17
+177 -17
+183 -215
+2 -20
+40 -12
+19 -374
+150 -373
+71 -590
+40 -46
+91 -189
+113 -388
+183 -373
+157 -373
+62 -239
+21 -384
+19 -377
+88 -3
+61 -6
+62 -1
+61 -2
+62 -1
+11 -117
+84 -0
+88 -28
+47 -0
+32 -66
+92 -293
+81 -220
+16 -220
+19 -55
+65 -20
+53 -2
+100 -2
+54 -2
+74 -2
+47 -8
+101 -221
+116 -183
+142 -188
+24 -16

+31 -223
+2 -0
+50 -13
+37 -38
+20 -10
+32 -48
+67 -1

+15 -8
+13 -0
+17 -1
+5 -1
+7 -2
+2 -0
+9 -1
+22 -41

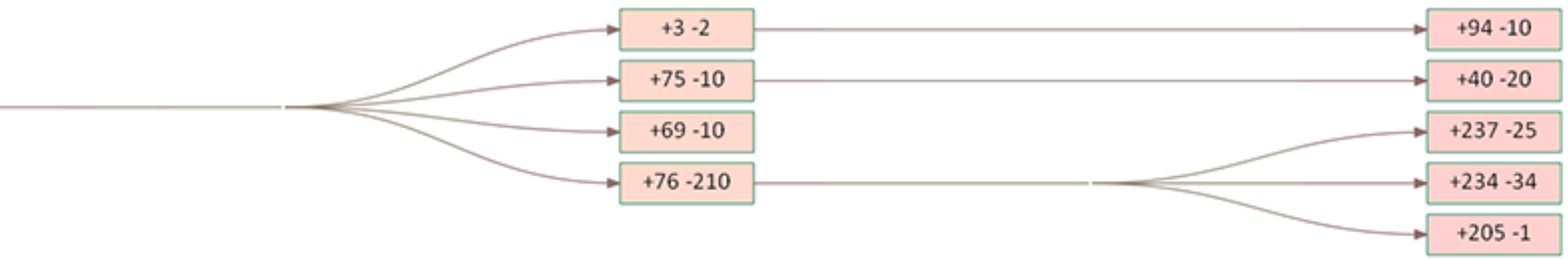
+8 -3
+8 -205

LEVEL 6 TEST CASES

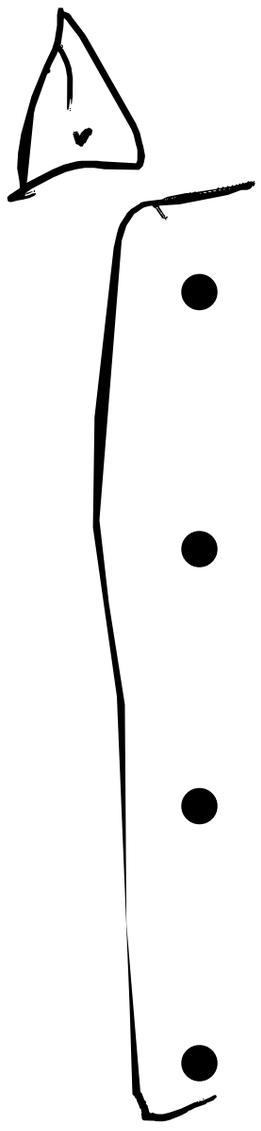
+3 -2
+75 -10
+66 -10
+76 -210
+54 -10
+40 -20
+237 -25
+234 -34
+205 -1

VALUES CORRESPOND TO THE NUMBER OF NEW STATE TRANSITIONS (TUPLES) APPEARING IN THE ISOLATED TEST CASE, COMPARED TO PARENT.

LEVEL 6 TEST CASES



Estrategias de mutación de AFL



- **Walking bit & byte flips**: Flipping un bit, dos bits, etc.
- **Simple arithmetics**: incrementar o decrementar valores enteros existents en el input
- **Known integers**: e.g., -1, 256, 1024, MAX_INT-1, MAX_INT
- **Stacked tweaks**: borrar, duplicar, insertar bloques

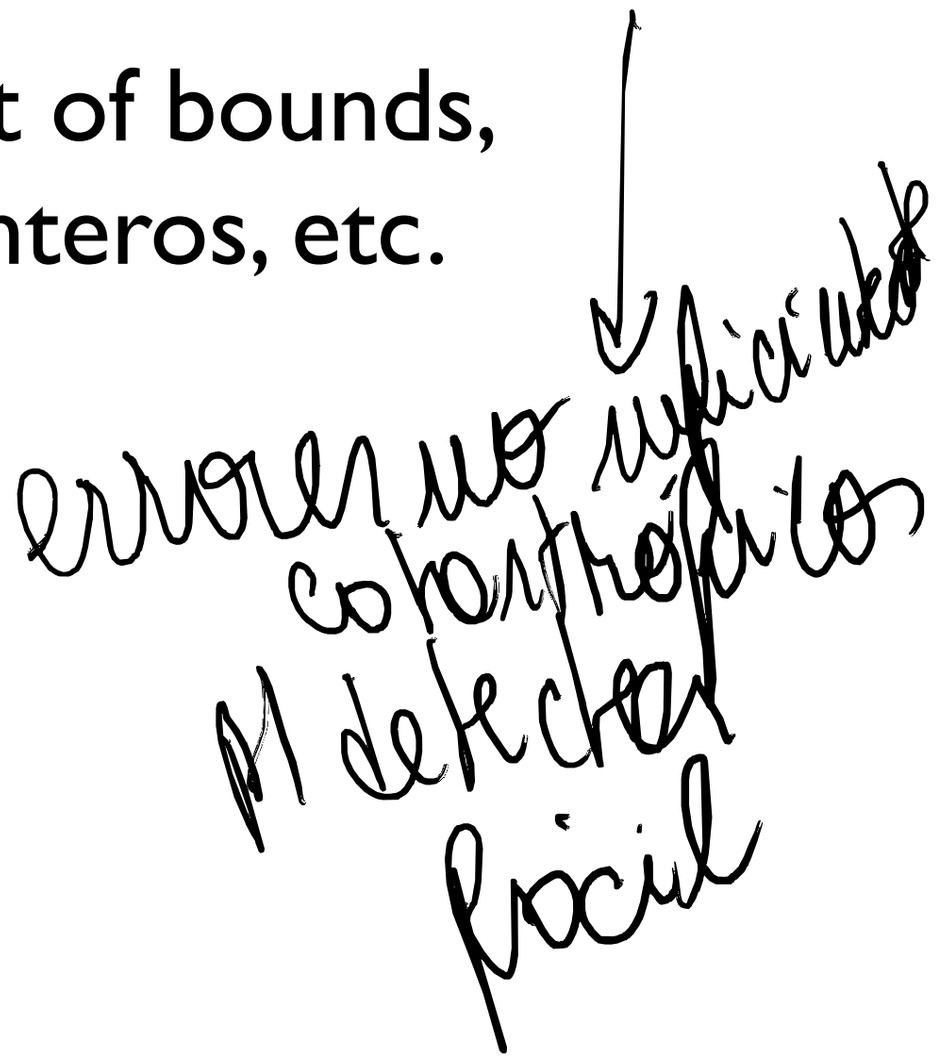
se sabe que son efectivos p/ vulnerabilidades típicas en C

AFL Strategies

- https://lcamtuf.blogspot.com.ar/2014/08/binary-fuzzing-strategies-what-works.html
- Las estrategias de mutación de inputs de AFL son operadores de mutación que provienen de la comunidad de seguridad (no de la comunidad de lenguajes de programación).
- Entonces, en su mayoría son manipulaciones de bits.

Más Allá de los Crashes

- Un programa crashea únicamente si hay un problema serio
  - Hangs, Segmentation fault, assertion failure, etc.
- Pero puede sobrevivir a defectos **latentes**
  - Memory Leaks, Index Out of bounds, Incorrecto Manejo de Punteros, etc.



Sanitizers

- Son frameworks que instrumentan un programa para realizar chequeos en tiempo de ejecución
- Existen distintos sanitizers para C/C++
- https://github.com/google/sanitizers

Example sanitizers

AddressSanitizer (ASan)

- Framework de Instrumentación para detectar múltiples clases de corrupción del manejo de Memoria
  - Heap/stack buffer overflows, use-after-free bugs
- Se puede combinar con cualquier Fuzzer siempre y cuando podamos instrumentar el programa

Another example

Undefined Behaviour Sanitizier (UBSan)

- Detecta en rutime operaciones cuya semántica no está bien descripta en C/C++
- Ejemplos:
  - Oversized Shift Amounts
  - Dereferencing a NULL Pointer
  - Violating Type Rules
  - Use of an uninitialized variable
  - Signed integer overflow

Memory Sanitizier (MSan)

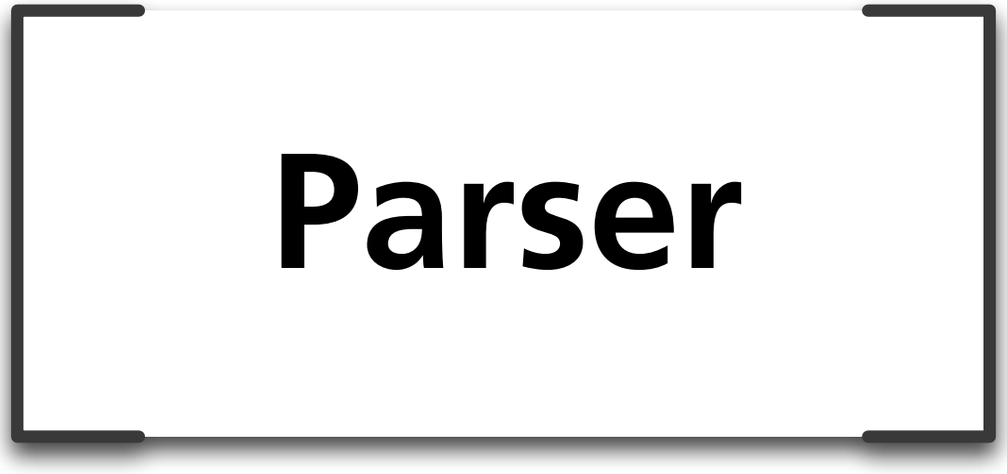
- Detecta un-initialised reads en el heap
- Usualmente introduce un alentecimiento de 3x

Leak Sanitizier (LSan)

- Detecta memory leaks

Mutation-Test fuzzing

- Supongamos ahora que queremos fuzzear el Intérprete de Javascript de un browser
- Ej: SpiderMonkey
- Este Intérprete necesita compilar y ejecutar el programa Javascript



Mutation-Test fuzzing

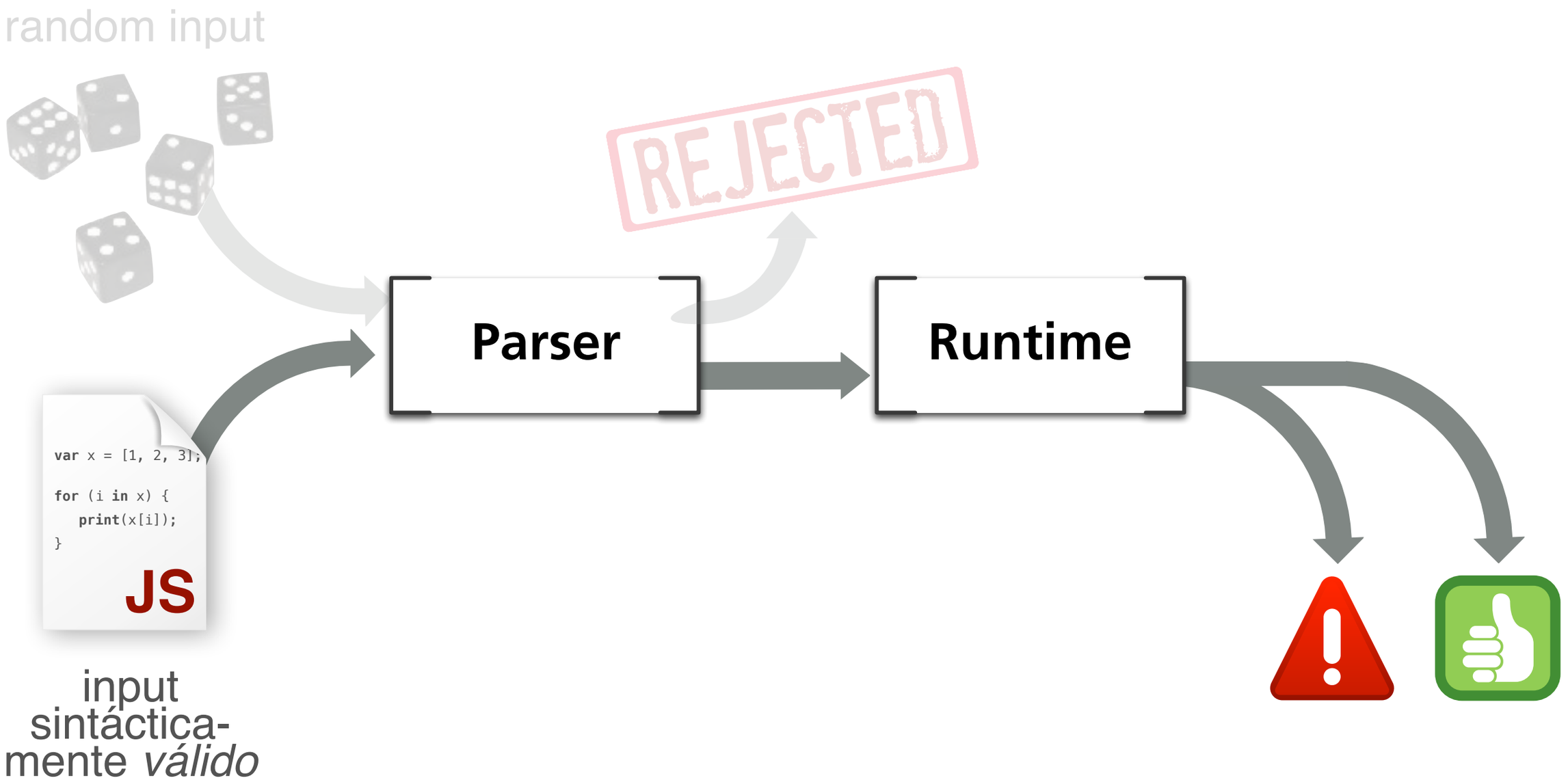
random input

REJECTED

Parser

Runtime

Mutation-Test fuzzing



Mutation-based fuzzing

- Los inputs iniciales (seed) nos ayudan a tener versiones alternativas que sobrevivan la etapa de validación.
- No obstante, ¿cómo aseguramos que **TODOS** los inputs generados van a sobrevivir la etapa de validación?

Con práctica podemos chequear inputs válidos.

Gramática JavaScript

If Statement

IfStatement[full] ⇒
  `if` *ParenthesizedExpression* *Statement*[full]
| **`if`** *ParenthesizedExpression* *Statement*[noShortIf] **`else`** *Statement*[full]
IfStatement[noShortIf] ⇒ **`if`** *ParenthesizedExpression* *Statement*[noShortIf] **`else`** *Statement*[noShortIf]

Switch Statement

SwitchStatement ⇒
  `switch` *ParenthesizedExpression* **`{ }`**
| **`switch`** *ParenthesizedExpression* **`{`** *CaseGroups* *LastCaseGroup* **`}`**
CaseGroups ⇒
  «empty»
| *CaseGroups* *CaseGroup*
CaseGroup ⇒ *CaseGuards* *BlockStatementsPrefix*
LastCaseGroup ⇒ *CaseGuards* *BlockStatements*
CaseGuards ⇒
  CaseGuard
| *CaseGuards* *CaseGuard*
CaseGuard ⇒

Gramáticas

- Una gramática es un conjunto de reglas "L->R" donde
  - L es un simbolo no terminal
  - R es una cadena de símbolos terminales y no terminales

Una Gramática para árboles

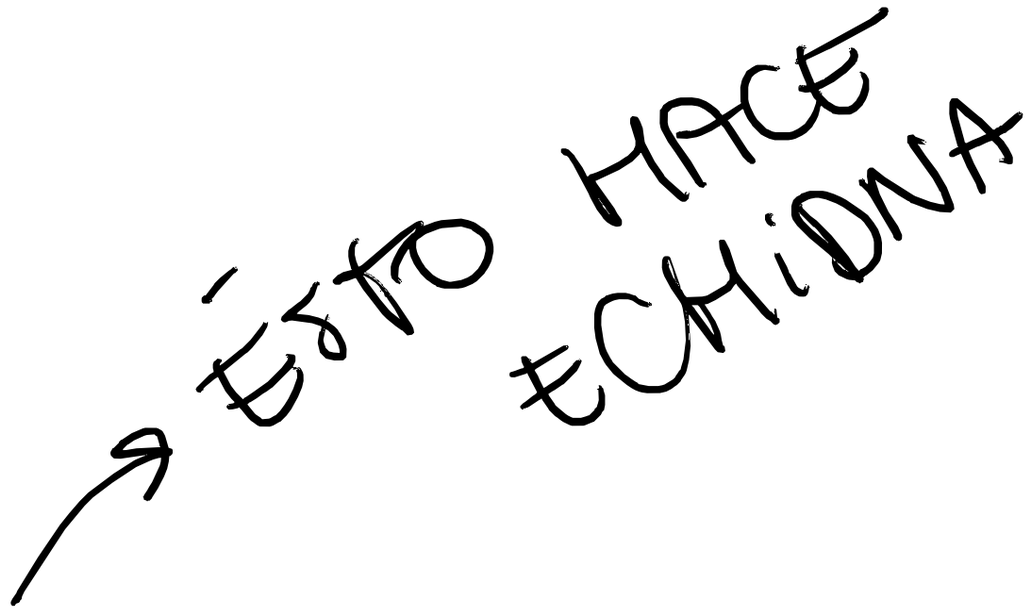
- Una gramática para árboles
  - $TREE -> “x”
  - $TREE -> “($TREE,$TREE)”
- Son cadenas posibles…
  - x ?
  - (x,x) ?
  - ((x,x),x) ?
  - (x) ?
  - (x,) ?

Gramáticas para Expresiones Aritméticas

```
$START –>  "$EXPR"

$EXPR –> "$EXPR + $TERM"
$EXPR –>"$EXPR – $TERM"
$EXPR –> “$TERM"
$TERM –> “$TERM * $FACTOR”
$TERM –> “$TERM / $FACTOR”
$TERM –> “$FACTOR”

$FACTOR –> “+$FACTOR"
$FACTOR –> “–$FACTOR"
$FACTOR –> "($EXPR)"
$FACTOR –> "$INTEGER"
$FACTOR –> "$INTEGER.$INTEGER"

$INTEGER –> "$INTEGER$DIGIT"
$INTEGER –> “0” | … | “9”
```

$EXPR
$EXPR – $TERM
$EXPR + $TERM – $TERM
$EXPR + $TERM * $FACTOR – $TERM
$TERM + $TERM * $FACTOR – $TERM
$TERM + $TERM * –$FACTOR – $TERM
$FACTOR + $TERM * –$FACTOR – $TERM
–$FACTOR + $TERM * –$FACTOR – $TERM
—$FACTOR + $TERM * –$FACTOR – $TERM
—$FACTOR + $FACTOR * –$FACTOR – $TERM
——$FACTOR + $FACTOR * –$FACTOR – $TERM
——$FACTOR + $FACTOR * –$FACTOR – $FACTOR
——+$FACTOR + $FACTOR * –$FACTOR – $FACTOR
——+–$FA
——+–$IN
——+–$D
——+–2 +
——+–2 + $INTEGER.$INTEGER * –$FACTOR – $FACTOR
——+–2 + $INTEGER.$INTEGER * –+$FACTOR – $FACTOR
——+–2 + $INTEGER.$INTEGER * –+$INTEGER – $FACTOR
——+–2 + $DIGIT.$INTEGER * –+$INTEGER – $FACTOR
——+–2 + 3.$INTEGER * –+$INTEGER – $FACTOR
——+–2 + 3.$INTEGER * –+$INTEGER – +$FACTOR
——+–2 + 3.$INTEGER * –+$INTEGER – +$INTEGER
——+–2 + 3.$DIGIT * –+$INTEGER – +$INTEGER
——+–2 + 3.5 * –+$INTEGER – +$INTEGER
——+–2 + 3.5 * –+$DIGIT – +$INTEGER
——+–2 + 3.5 * –+1 – +$INTEGER
——+–2 + 3.5 * –+1 – +$DIGIT
——+–2 + 3.5 * –+1 – +5

——+–2 + 3.5 * –+1 – +5



Grammar-based Input Generation

- Un programa puede aceptar inputs con una estructura determinada (expresiones aritméticas, direcciones de mail, páginas web, etc.)
- Podemos usar una gramática para generar aleatoriamente inputs válidos que espera el programa
- Es más efectivo que hacer random testing "clásico".

```
$DOCUMENT -> “$DOCTYPE$HTML”

$DOCTYPE -> '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"\n' + \
    '"http://www.w3.org/TR/html4/strict.dtd">\n'

$HTML  -> "<HTML>$HEAD$BODY</HTML>\n"

$HEAD  -> "<HEAD>$TITLE</HEAD>\n"
$TITLE -> "<TITLE>A generated document</TITLE>\n"

$BODY -> "<BODY>$DIVS</BODY>\n"

$DIVS -> "$DIV"
$DIVS -> "$DIV\n$DIVS"

$DIV -> "$HEADER\n$LIST"

$HEADER -> "<H1>A header.</H1>"
$HEADER -> "<H1>Another header.</H1>"

$LIST -> "<UL>$ITEMS</UL>"
$LIST -> "<OL>$ITEMS</OL>"

$ITEMS -> "$ITEM"
$ITEMS -> "$ITEM$ITEMS"

$ITEM -> "<LI>$TEXT</LI>\n"

$TEXT -> "An item"
$TEXT -> "Another item"
```

Random HTML

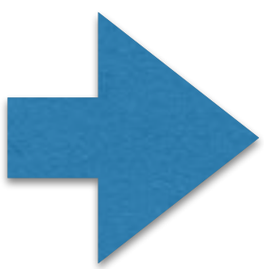
Grammar-based Input Generation

- Un programa puede aceptar inputs con una estructura determinada (expresiones aritméticas, direcciones de mail, páginas web, etc.)
- Podemos usar una gramática para generar aleatoriamente inputs válidos que espera el programa
- Es más efectivo que hacer random testing "clásico".

Grammar-based Input Generation

```
$START ->  "$EXPR"

$EXPR -> "$EXPR + $TERM"
$EXPR ->"$EXPR – $TERM"
$EXPR -> “$TERM"
$TERM -> “$TERM * $FACTOR”
$TERM -> “$TERM / $FACTOR”
$TERM -> “$FACTOR”

$FACTOR -> “+$FACTOR"
$FACTOR -> “–$FACTOR"
$FACTOR -> "($EXPR)"
$FACTOR -> "$INTEGER"
$FACTOR -> "$INTEGER.$INTEGER"

$INTEGER -> "$INTEGER$DIGIT"
$INTEGER -> “0” | … | “9”
```

Gramática



Random Grammar-based Generator

——+–2 + 3.5 * –+1 – +5

Inputs Válidos

——+–2 + 3.5 * –+1 – +5


```
public static void Main()
{
    byte[] data = new byte[10
    TcpClient server;
    try{
        server = new TcpClie
    }catch (SocketException
        Console.WriteLine("
        return;
    }
    NetworkStream ns = se
```

Programa bajo Test

Crashes
Hangs

Ejemplo de juguete

Un generador de inputs basado en gramáticas

```
MAXSYMBOLS = 5

term = "$START"
while term has no terminal symbols
   rule = choose random grammar rule
   new_term = apply rule to term
   if number_of_non_terminals(new_term) < MAXSYMBOLS
    term = new_term

return term
```

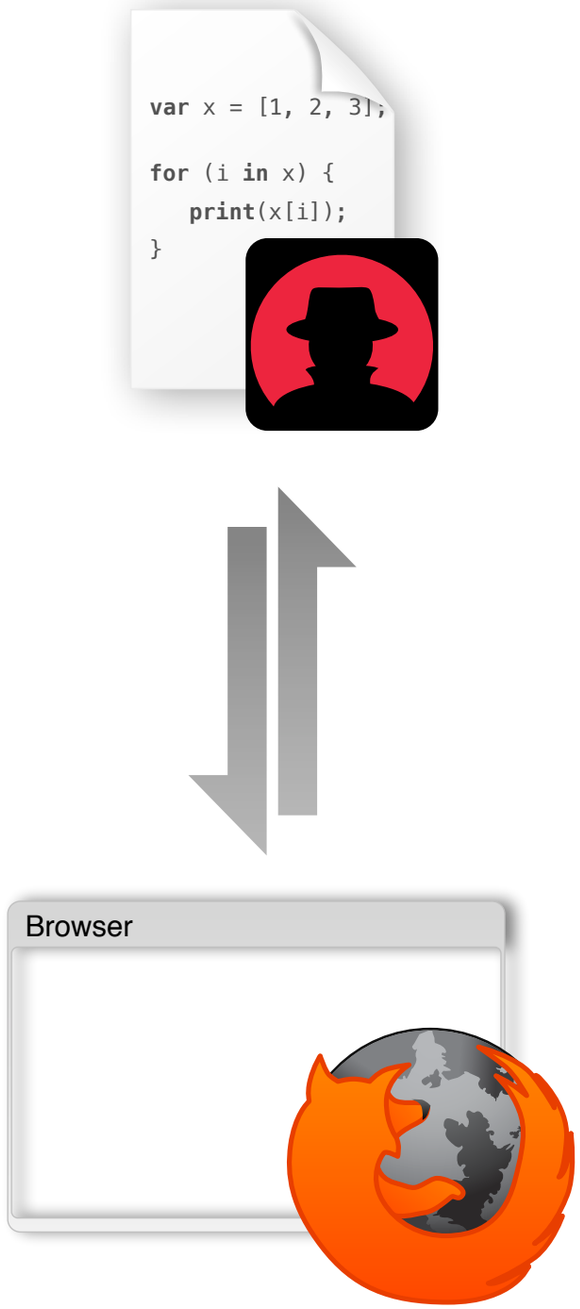
Grammar-based Input Generation

- Existen distintas herramientas que generan inputs sobre gramáticas:
  - LangFuzz: programas JS
  - DomFuzz: documentos Dom
  - JSFuzz: programas JS
  - CSmith: programas C
  - XMLMate: documentos XML

Fuzzing de Gramáticas

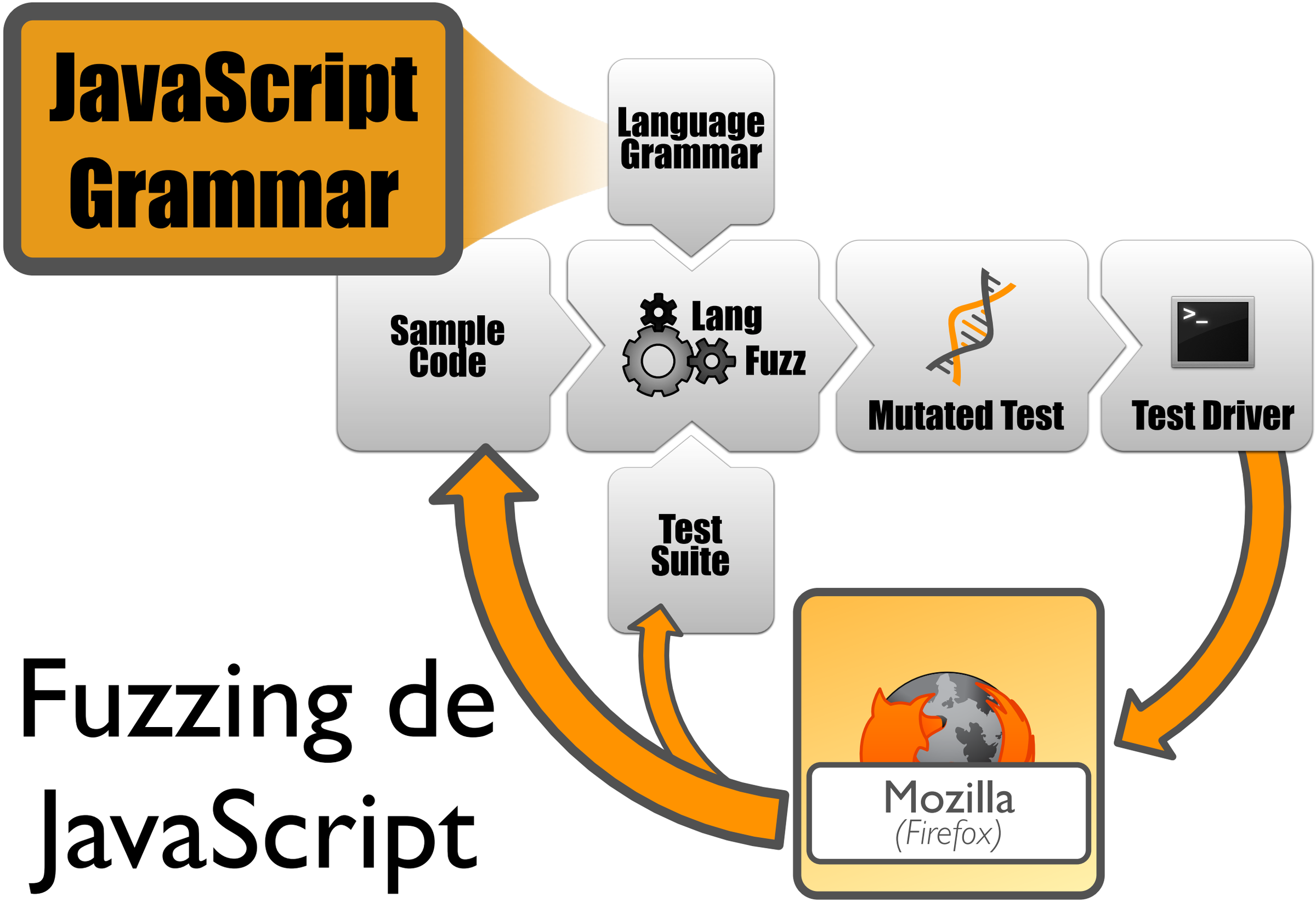
- LangFuzz es un Fuzz Tester que usa una *gramática* para generar inputs
- *Utiliza Inputs Válidos* para crear nuevos inputs

Gramática: JavaScript



- Si un atacante toma el control del Intérprete *JavaScript*, toma el control del ***browser entero***

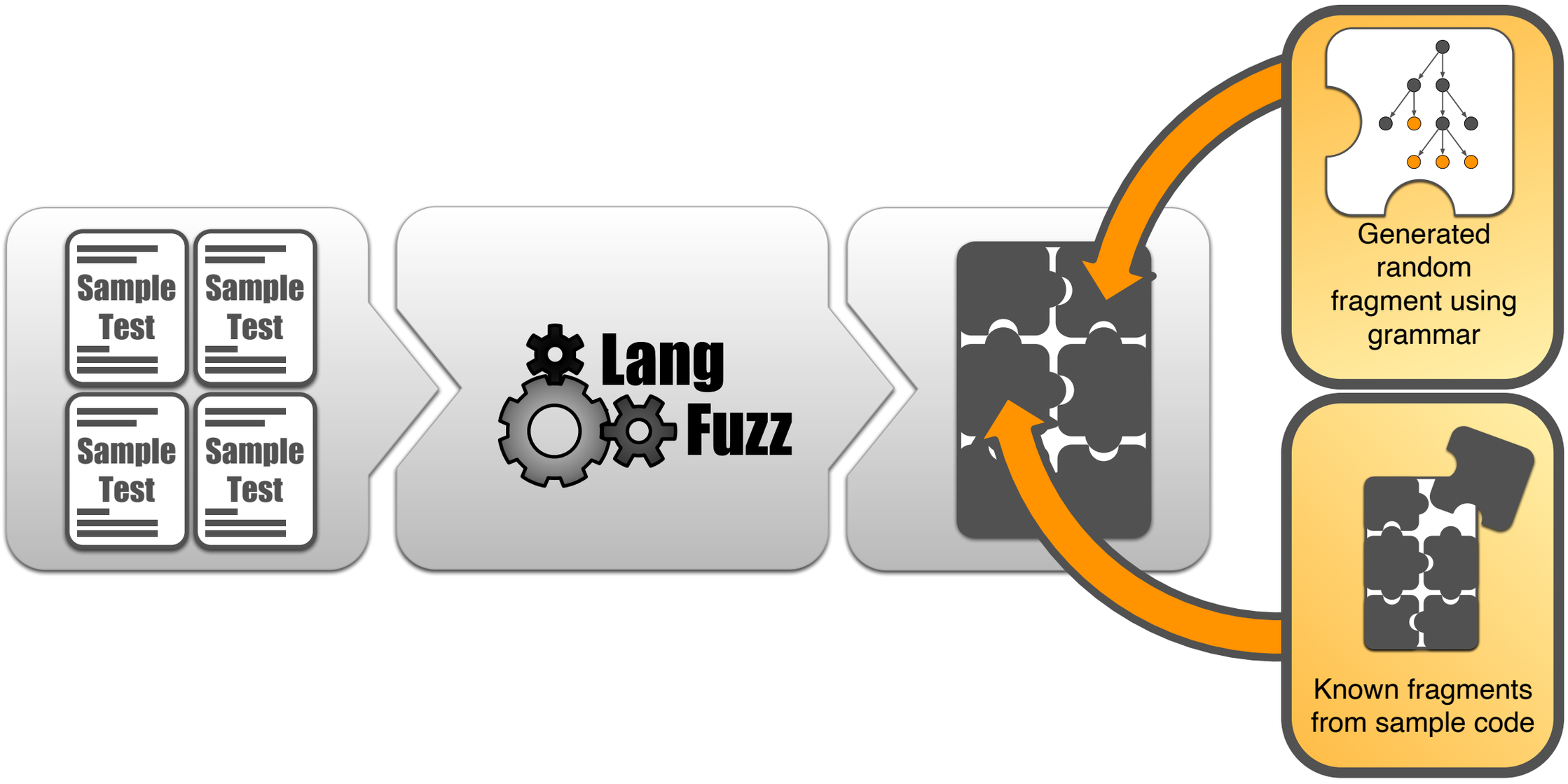
Fuzzing de JavaScript



LangFuzz

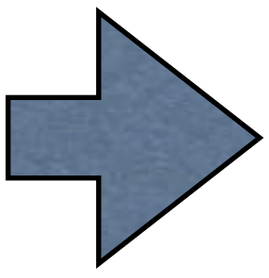
- Genera aleatoriamente programas JavaScript válidos a partir de la gramática del lenguaje JavaScript
- Extiende la gramática con nuevos terminales que son fragmentos de código JavaScript que ocasionaron vulnerabilidades en el pasado
- Idea: atacar "fixes" incompletos

Usando Fragmentos de Código JavaScript



Un Input Generado con LangFuzz

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



LangFuzz (Feb 2023)

En uso en Mozilla desde 2011,

2397 bugs en el engine JS.

En Google desde 2014, **~27,000 bugs en Google** encontrados.

43 Chrome Security Bug Bounties hasta **hoy**

www.mozilla.org/security/bug-bounty/

Bug Bounty Program

Introduction

The Mozilla Security Bug Bounty Program is designed to encourage security research in Mozilla software and to reward those who help us create the safest Internet clients in existence.

Many thanks to Linspire and Mark Shuttleworth, who provided start-up funding for this endeavor.

Mozilla has paid out over 1.6 million dollars in bounties to our various researchers!

Mozilla manages two different bug bounty programs. One program focuses on Firefox and other client applications and one bounty program focuses on our web properties and services.

- Information on the Client Bug Bounty Program can be found here
- Information on the Web and Services Bug Bounty Program can be found here

Mozilla Security

Security Advisories

Known Vulnerabilities

Bug Bounty

Firefox Hall Of Fame

Mozilla Web and Services Hall Of Fame

Security Blog

JS FunFuzz

```
function makeBranchUnstableLoop(d, b) {
  var reps = loopCount();
  var v = uniqueVarName();
  var mod = loopModulo();
  var target = rnd(mod);
  return forLoopHead(d, b, v, reps) + " { " +
    "if (" + v + " % " + mod + " == " + target + ") { "
        + makeStatement(d - 2, b) + " } " +
    "else { "
        + makeStatement(d - 2, b) + " } " +
    "}";
}

function makeTypeUnstableLoop(d, b) {
  var a = makeMixedTypeArray(d, b);
  var v = makeNewId(d, b);
  var bv = b.concat([v]);
  return "for each (let " + v + " in " + a + ") { "
         + makeStatement(d - 2, bv) +
        "}";
}


function makeFunOnCallChain(d, b) {
  var s = "arguments.callee";
  while (rnd(2))
    s += ".caller";
  return s;
}
```

- Generador de JS escrito por Jesse Ruderman
- Soporta la mayoría del standard ECMA y extensiones de Mozilla
- 2869 bugs encontrados desde 2006

https://github.com/MozillaSecurity/funfuzz/tree/master/js/jsfunfuzz

DOMFuzz

```
function randomCSSAnimation() {
  var animName = Random.pick(fuzzValues.names);
  var s = "@keyframes " + animName + " { ";

  for (var j = 0; j < 3; ++j) {
    s += randomKeyframePoints();
    s += " { ";
    for (var i = 0; i < 3; ++i) {
      var decl = randomDeclaration();
      // no !important inside animations
      s += decl.prop + ": " + decl.value + "; ";
    }
    s += "} ";
  }

  s += "} ";

  if (rnd(4)) {
    var duration = Random.pick(fuzzValues.durations);
    s += randomSelector()
        + " { animation-name: " + animName
        + "; animation-duration: " + duration + "; }";
  }

  return s;
}

function randomKeyframePoints() {
  return Random.index(["from", "to", "0%", "20%", "40%",
                       "60%", "80%", "100%", "20%, 50%"]);
}
```

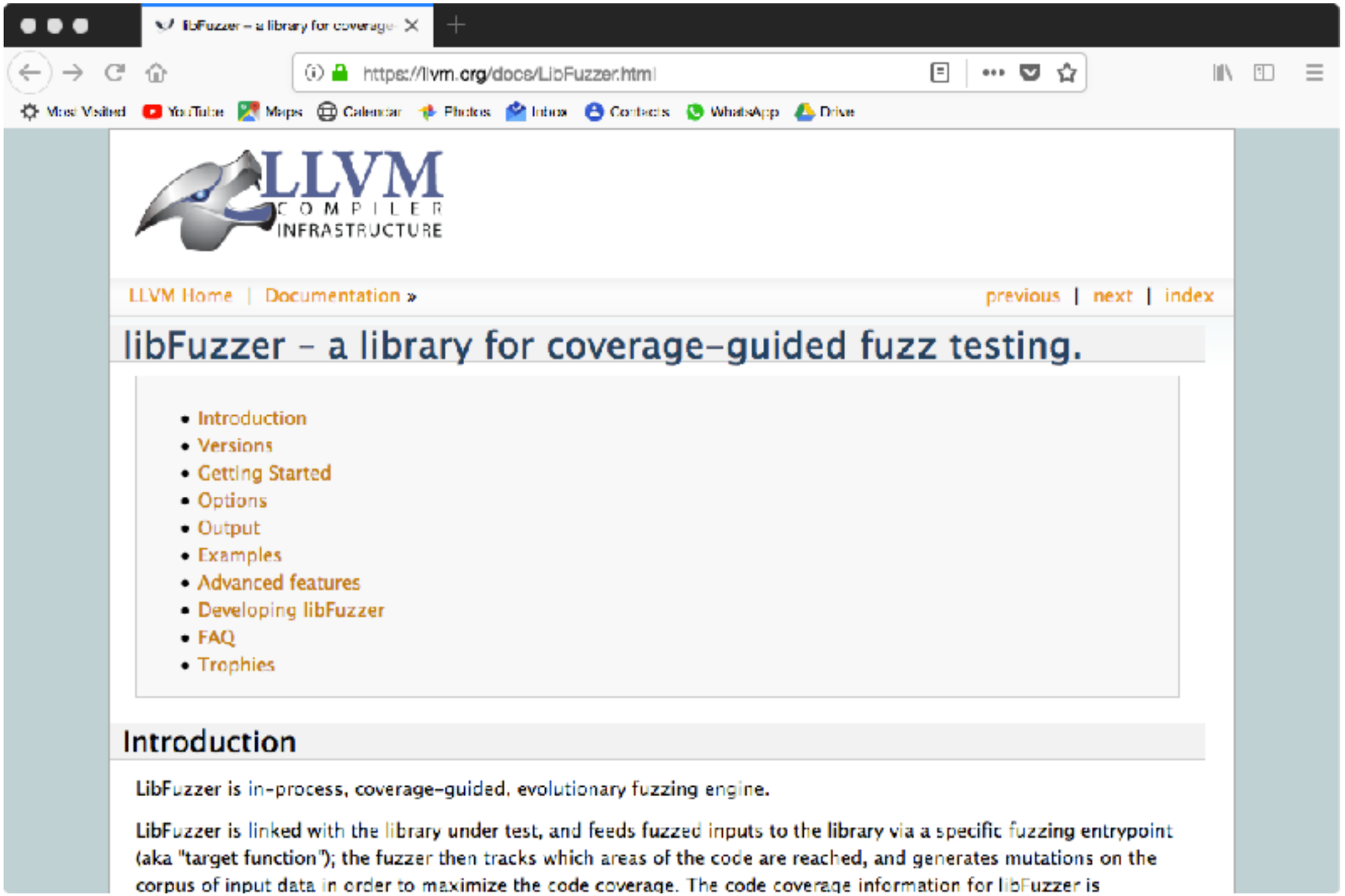
- Generador/Mutador de DOM escrito por Jesse Ruderman
- Arquitectura modular: diferentes módulos para distintos tipos de DOM, e.g. CSS, Events, WebAudio, SVG, Unicode ...

https://github.com/MozillaSecurity/domfuzz

Grammar Fuzzing

- La mayoría de los Fuzzers para gramáticas aplican variaciones de **Grammar** Testing (Bounded **Exhaustive** Testing)
- LangFuzz = Los Code Fragments son Terminales
- Como la gramática resultante es muy grande, se elige aleatoriamente la producción a aplicar

LibFuzzer


libFuzzer – a library for coverage
https://llvm.org/docs/LibFuzzer.html
Most Visited
YouTube
Maps
Calendar
Photos
Inbox
Contacts
WhatsApp
Drive
LLVM
COMPILER
INFRASTRUCTURE
LLVM Home | Documentation »
previous | next | index
libFuzzer – a library for coverage-guided fuzz testing.
Introduction
Versions
Getting Started
Options
Output
Examples
Advanced features
Developing libFuzzer
FAQ
Trophies
Introduction
LibFuzzer is in-process, coverage-guided, evolutionary fuzzing engine.
LibFuzzer is linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entrypoint (aka "target function"); the fuzzer then tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage. The code coverage information for libFuzzer is

LibFuzzer

- AFL: Fuzzea solo sistemas enteros
- LibFuzzer: permite fuzzear una librería
  - Necesita implementar una función que transforma un arreglo de bytes en un input válido de la librería

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;  // Non-zero return values are reserved for future use.
}
```

Propiedades del

LibFuzzer

- El código debe poder tolerar cualquier input de data
- No debe haber llamados a exit()
- No debe utilizar threads/no determinismo
- Tan rápido como se pueda (el target)
- No modificar información global (static)

Corpus Distillation

Google fuzzed Adobe Flash in 2011:

„What does **corpus distillation look like at Google scale?** Turns out we have a large index of the web, so we cranked through **20 terabytes of SWF file downloads** followed by **1 week of run time on 2,000 CPU cores** to calculate the **minimal set of about 20,000 files**. Finally, those same **2,000 cores plus 3 more weeks** of runtime were put to good work **mutating the files** in the minimal set (bitflipping, etc.) and generating crash cases. “

The initial run of the ongoing effort resulted in about **400 unique crash signatures**, which were logged as 106 individual security bugs following Adobe's initial triage.

- Source: https://security.googleblog.com/2011/08/fuzzing-at-scale.html

Dom Fuzzer

Google fuzzed the DOM of major browsers in 2017:

https://googleprojectzero.blogspot.co.at/2017/09/the-great-dom-fuzz-off-of-2017.html

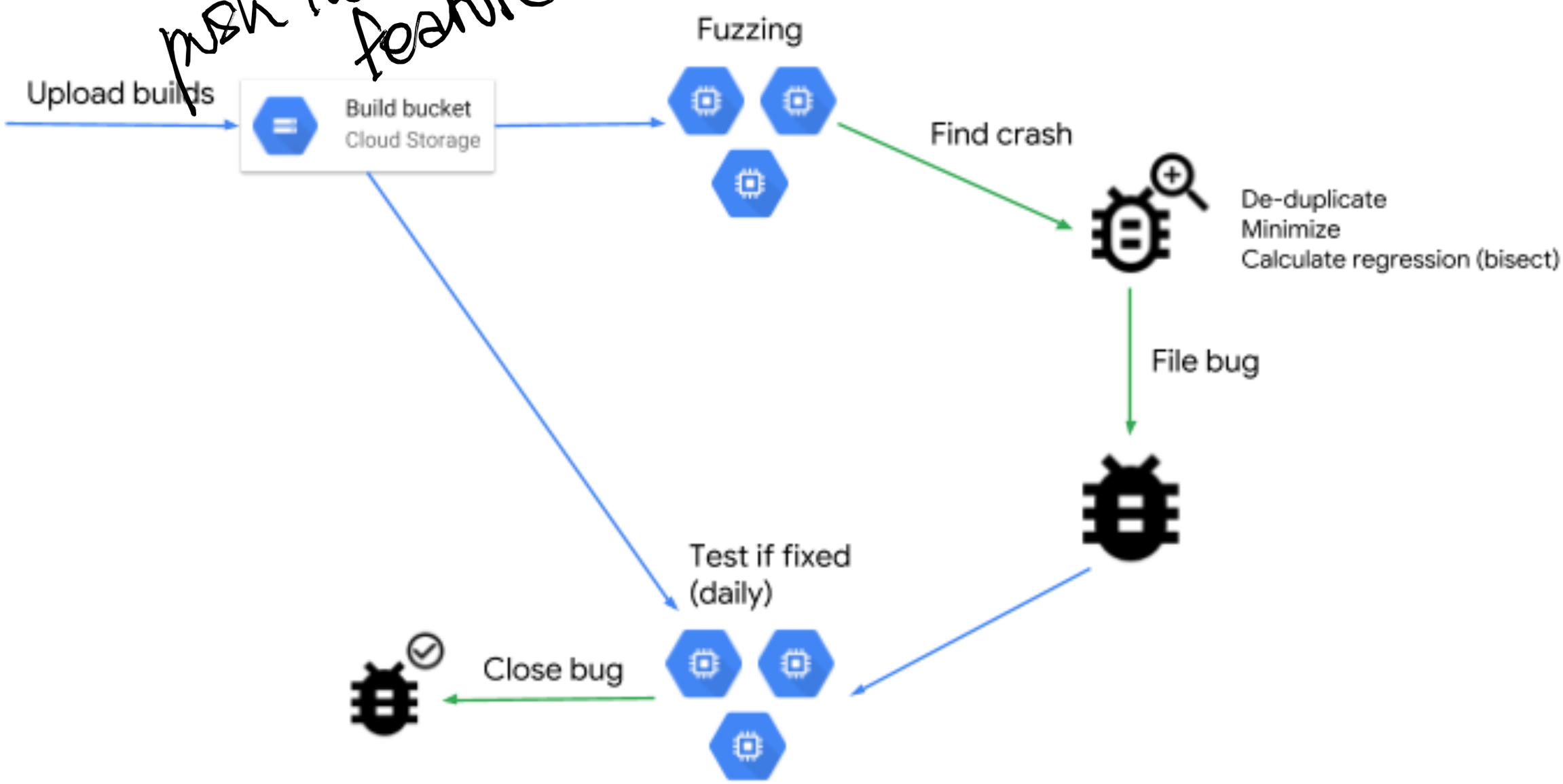
We tested 5 browsers with the highest market share: Google Chrome, Mozilla Firefox, Internet Explorer, Microsoft Edge and Apple Safari. We gave each browser approximately 100.000.000 iterations with the fuzzer and recorded the crashes. (If we fuzzed some browsers for longer than 100.000.000 iterations, only the bugs found within this number of iterations were counted in the results.) Running this number of iterations would take too long on a single machine and thus requires fuzzing at scale, but it is still well within the pay range of a determined attacker. For reference, it can be done for about $1k on Google Compute Engine given the smallest possible VM size, preemptable VMs (which I think work well for fuzzing jobs as they don't need to be up all the time) and 10 seconds per run.

ClusterFuzz

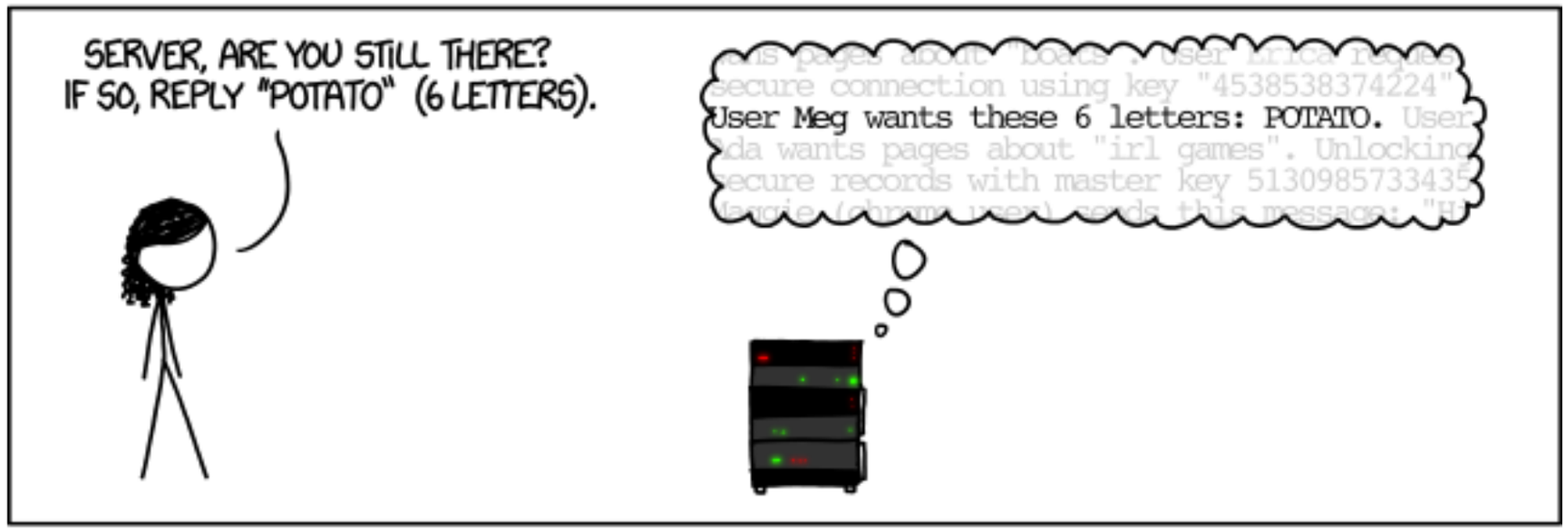
- Infraestructura para ejecutar fuzzing sobre software
- Google usa ClusterFuzz para fuzzear Chrome Browser
- Scalable. El cluster fuzz de Google's corre sobre 25,000 máquinas.
- Enero 2019, ClusterFuzz detectó ~16,000 bugs en Chrome

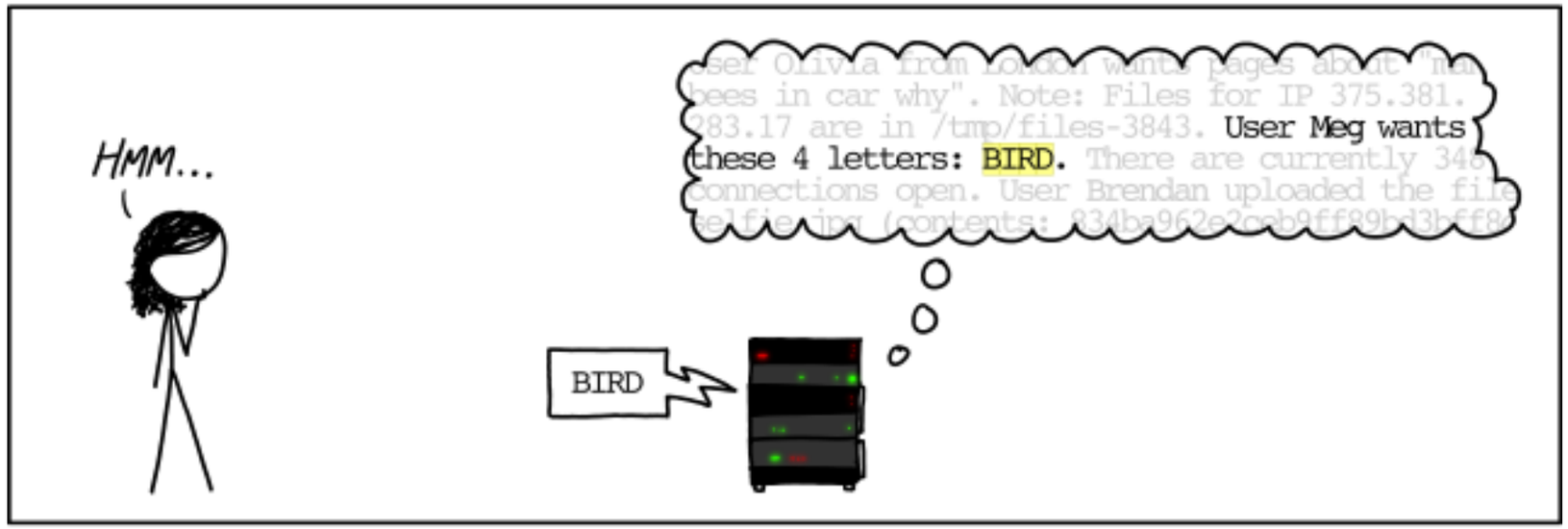
Integración continua de fuzzing.

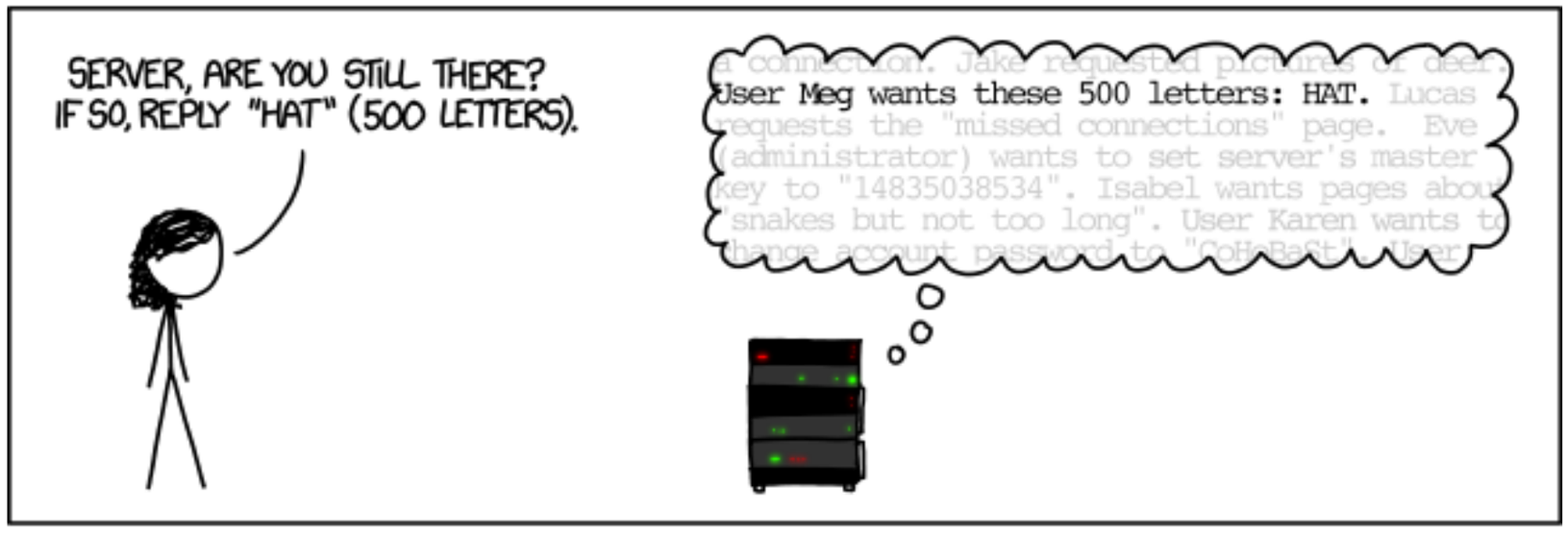
ClusterFuzz

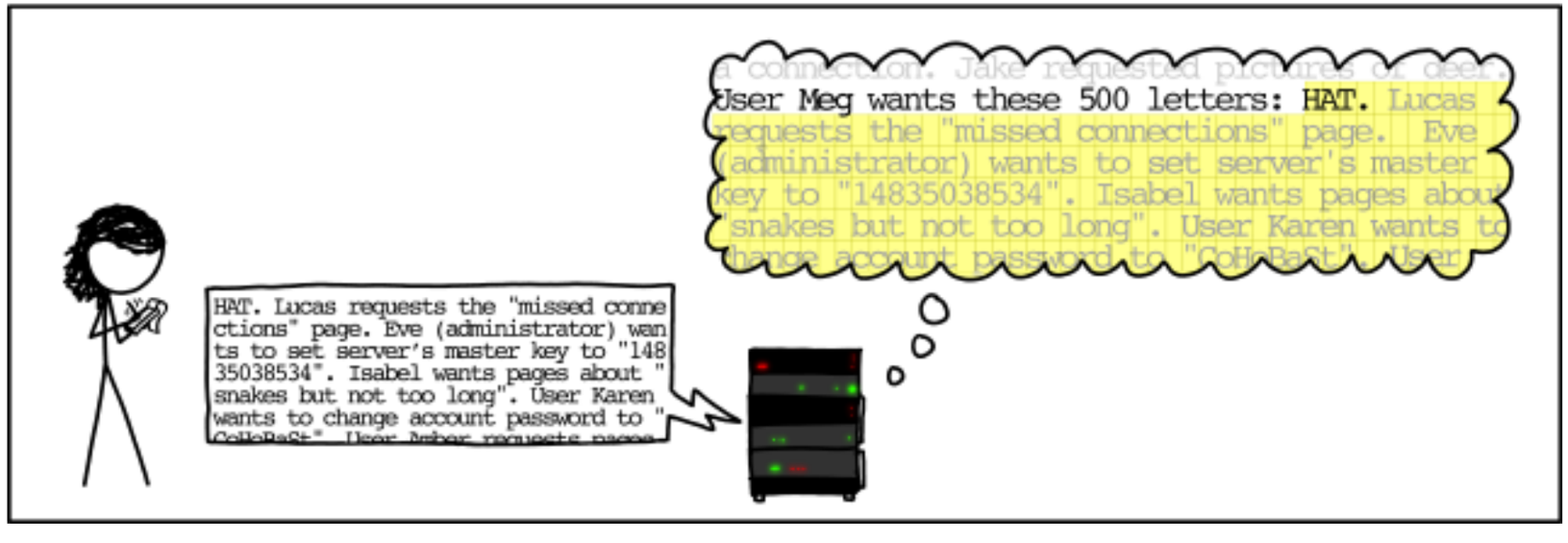


HOW THE HEARTBLEED BUG WORKS:




HMM...
BIRD
User Olivia from London wants pages about "mad bees in car why". Note: Files for IP 375.381.383.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 346 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2eab9ff89bd3bff8


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHeBaSt". User


User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHeBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "

Heartbleed

- Dic. 2013: primer release de AFL
- Nov 2014: primer release de LibFuzzer
- Heartbleed:
  - AFL: 6 horas
  - Libfuzzer: 10 segs

Heartbleed

- Heartbleed se introduce en Dic. 2011
- Recien se detecta en Marzo 2014
- Si era fácil de detectar por AFL, ¿por qué no se detectó?

Heartbleed

- OpenSSL era un proyecto bien financiando
- Las herramientas de fuzzing existían
- El fuzzing no lo efectuaban los developers, sino los security researchers

OSS-Fuzz - Continuous Fuzzing for Open Source Software

> *Status*: Stable. We are accepting applications from widely-used open source projects.

FAQ | Ideal Fuzzing Integration | New Project Guide | Reproducing Bugs | Projects | Projects Issue Tracker | Glossary

Create New Issue for questions or feedback about OSS-Fuzz.

Introduction

Fuzz testing is a well-known technique for uncovering various kinds of programming errors in software. Many of these detectable errors (e.g. buffer overflow) can have serious security implications.

We successfully deployed guided in-process fuzzing of Chrome components and found hundreds of security vulnerabilities and stability bugs. We now want to share the experience and the service with the open source community.

In cooperation with the Core Infrastructure Initiative, OSS-Fuzz aims to make common open source software more secure and stable by combining modern fuzzing techniques and scalable distributed execution.

At the first stage of the project we use libFuzzer with Sanitizers. More fuzzing engines will be added later. ClusterFuzz provides a distributed fuzzer execution environment and reporting.

Currently OSS-Fuzz supports C and C++ code (other languages supported by LLVM may work too).

OSS-Fuzz

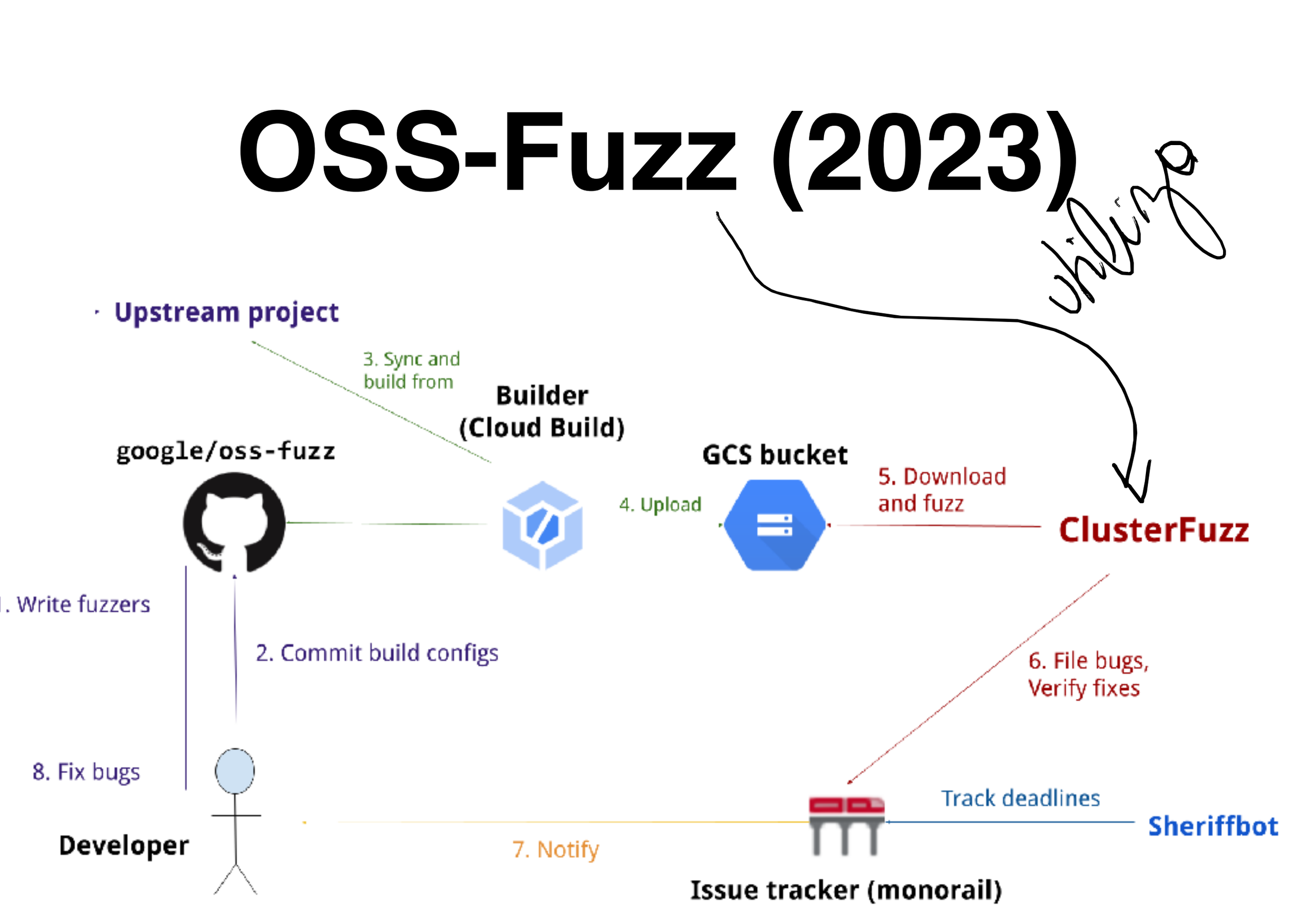
- Idea: suministrar infraestructura y tiempo de cómputo para fuzzear proyectos open-source importantes
- El proyecto Open-Source debe preparar el sistema para facilitar el fuzzing (libFuzzer/AFL)
- OSS-Fuzz fuzzea el proyecto y reporta vulnerabilidades y problemas detectados (ASan, UBSan, MSan)
- El proyecto Open-Source corrige el sistema

OSS-Fuzz: proyectos

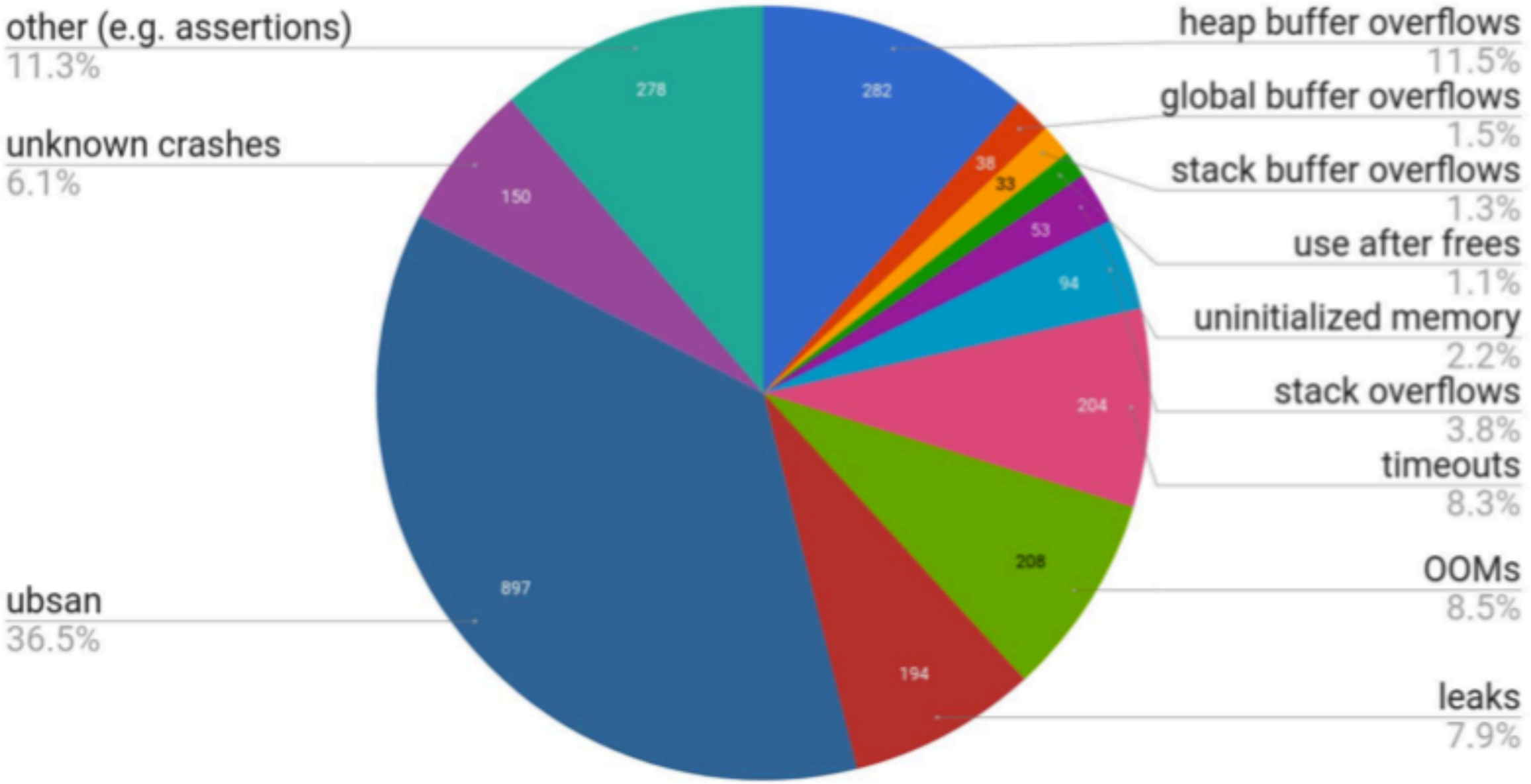
- Cpython2, cpython3
- Ffmpeg
- Coreutils
- Firefox
- Ghostscript
- Glib
- Golang
- Libmpeg2
- Libreoffice
- Llvm
- nodejs
- openssh
- openssl
- qt
- Sqlite3
- Syzkaller
- Tinyxml2
- Wget
- zlib
- …

https://github.com/google/oss-fuzz/tree/master/projects

OSS-Fuzz (2023)



OSS-Fuzz (>2000 bugs)



OSS-Fuzz: Ideal Integration

- Los tests targets son mantenidos por los programadores como parte de su codebase
- Existe un seed con buen coverage inicial
- Los fuzz targets son en general rápidos y no tienen OOM
- Provee un diccionario de fuzzing (constantes para ser usadas durante el fuzzing)

OSS-Fuzz: lifecycle

- AFL/Libfuzzer detectan una falla. Esta es de-duplicada contra los bugs existentes
- El input que produce la falla es minimizado
- El bug report es enviado privadamente a los programadores del proyecto
- Cada 24 horas el input es ejecutado contra la última versión del proyecto.
    - Fixed: el bug report se hace público 30 días despues del fix
    - Not-fixed: el bug report se hace público luego de 90 días

Bug report generado automáticamente

Issue 2445

Starred by 2 users

Status: Verified
Owner: ----
Closed: Jul 2
Cc: bi[redacted]@gmail.com
m[redacted].net
s[redacted]gmail.com
e[redacted]...@gmail.co[redacted]
Type: Bug-Security
ClusterFuzz
Stability-Memory-AddressSanitizer
Reproducible
ClusterFuzz-Verified
Engine-libfuzzer
OS-Linux
Proj-gdal
Reported-2017-07-01

gdal: Heap-buffer-overflow in PCIDSK::CBandInterleavedChannel::ReadBlock

Project Member Reported by monor...@clusterfuzz-external.iam.gserviceaccount.com, Jul 1

```
Detailed report: https://oss-fuzz.com/testcase?key=4766641567039488

Project: gdal
Fuzzer: libFuzzer_gdal_filesystem_fuzzer
Fuzz target binary: gdal_filesystem_fuzzer
Job Type: libfuzzer_asan_gdal
Platform Id: linux

Crash Type: Heap-buffer-overflow READ 4
Crash Address: 0x602000008855
Crash State:
  PCIDSK::CBandInterleavedChannel::ReadBlock
  PCIDSK2Band::IReadBlock
  GDALRasterBand::GetLockedBlockRef

Sanitizer: address (ASAN)

Recommended Security Severity: Medium

Regressed: https://oss-fuzz.com/revisions?job=libfuzzer_asan_gdal&range=201705291647:201705301648

Reproducer Testcase: https://oss-fuzz.com/download?testcase_id=4766641567039488
```

OSS-Fuzz

oss-fuzz | New issue | All issues | -status:WontFix,Duplicate -Infra

1 - 100 of 14194 Next › | List | Grid

ID	Type	Component	Status	Proj	Reported	Owner	Summary + Labels	Opened
17098	Bug	---	New	llvm	2019-09-09	---	llvm:llvm-itanium-demangle-fuzzer: Out-of-memory in llvm_llvm-itanium-demangle-fuzzer ClusterFuzz Reprocucible	20 hours ago
17091	Bug	---	New	llvm	2019-09-09	---	llvm:llvm-opt-fuzzer--x86_64-strength_reduce: ASSERT: !BaseRegs.empty() && "1*reg => reg, should not be needed." ClusterFuzz Reproducible	31 hours ago
17067	Build-Failure	---	New	iroha	---	---	iroha: Coverage build failure	2 days ago
17066	Build-Failure	---	Verified	uwebsockets	---	---	uwebsockets: Fuzzing build failure	2 days ago
17065	Build-Failure	---	New	php	---	---	php: Fuzzing build failure	2 days ago
17064	Build-Failure	---	Verified	perfetto	---	---	perfetto: Fuzzing build failure	2 days ago
17063	Build-Failure	---	New	openvswitch	---	---	openvswitch: Fuzzing build failure	2 days ago
17062	Build-Failure	---	New	cryptofuzz	---	---	cryptofuzz: Fuzzing build failure	2 days ago
17049	Bug	---	New	llvm	2019-09-07	---	llvm:llvm-itanium-demangle-fuzzer: ASSERT: Parser->TemplateParams.size() >= OldNumTemplateParamLists ClusterFuzz Reproducible	3 days ago
17048	Build-Failure	---	Verified	minizinc	---	---	minizinc: Fuzzing build failure	3 days ago
17047	Build-Failure	---	New	grpc	---	---	grpc: Fuzzing build failure	3 days ago
17046	Build-Failure	---	New	chakra	---	---	chakra: Fuzzing build failure	3 days ago

"As of August 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects."

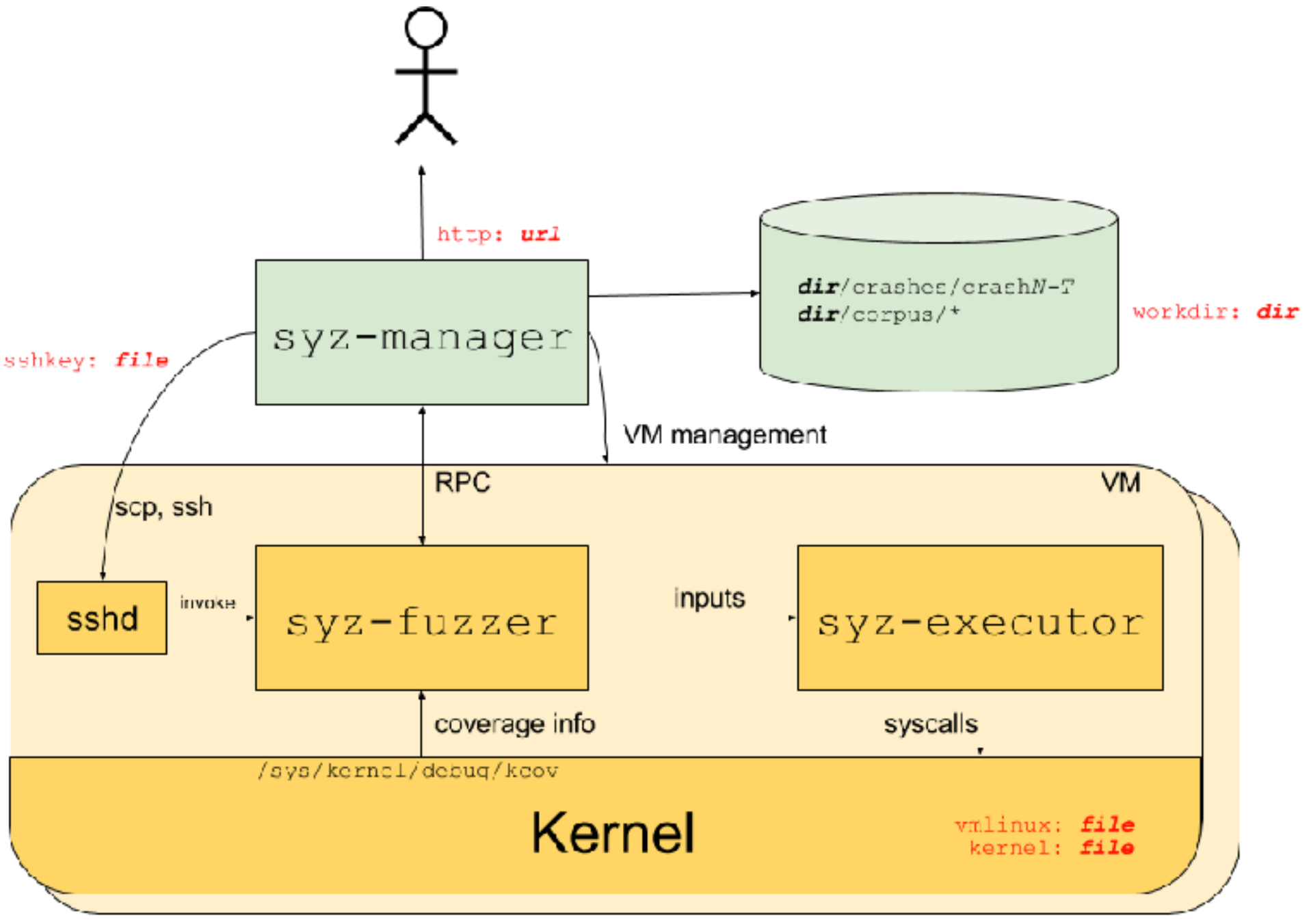
Patch reward programs

- Hasta $20,000: ideal integration con OSS-Fuzz
- $10,000 por bug-fixes complicados de alto impacto en vulnerabilidad
- $5,000 por bug-fixes moderados
- $1,337 por submitir fixes de complejidad baja
- $500 "one-liner specials" para fixes sencillos con impacto en seguridad

Syzkaller

- AFL/Libfuzzer fuzzean código de una aplicación
- ¿Qué pasa si la vulnerabilidad no reside en un defecto en el programa, sino en el sistema operativo (i.e. kernel)?
- Syzkaller es un fuzzer especializado para hacer coverage-based fuzzing de kernel code
  - ¿Cómo medimos cobertura en código kernel?
  - ¿Cómo instrumentamos el sistema operativo?
  - Usa VMs para invocar al sistema operativo objetivo

Syzkaller


http: url
syz-manager
dir/crashes/crashN-T
dir/corpus/*
workdir: dir
sshkey: file
VM management
RPC
VM
scp, ssh
sshd
invoke
syz-fuzzer
inputs
syz-executor
coverage info
syscalls
/sys/kernel/debug/kcov
Kernel
vmlinux: file
kernel: file

OneFuzz

microsoft/**onefuzz**

A self-hosted Fuzzing-As-A-Service platform

28 Contributors

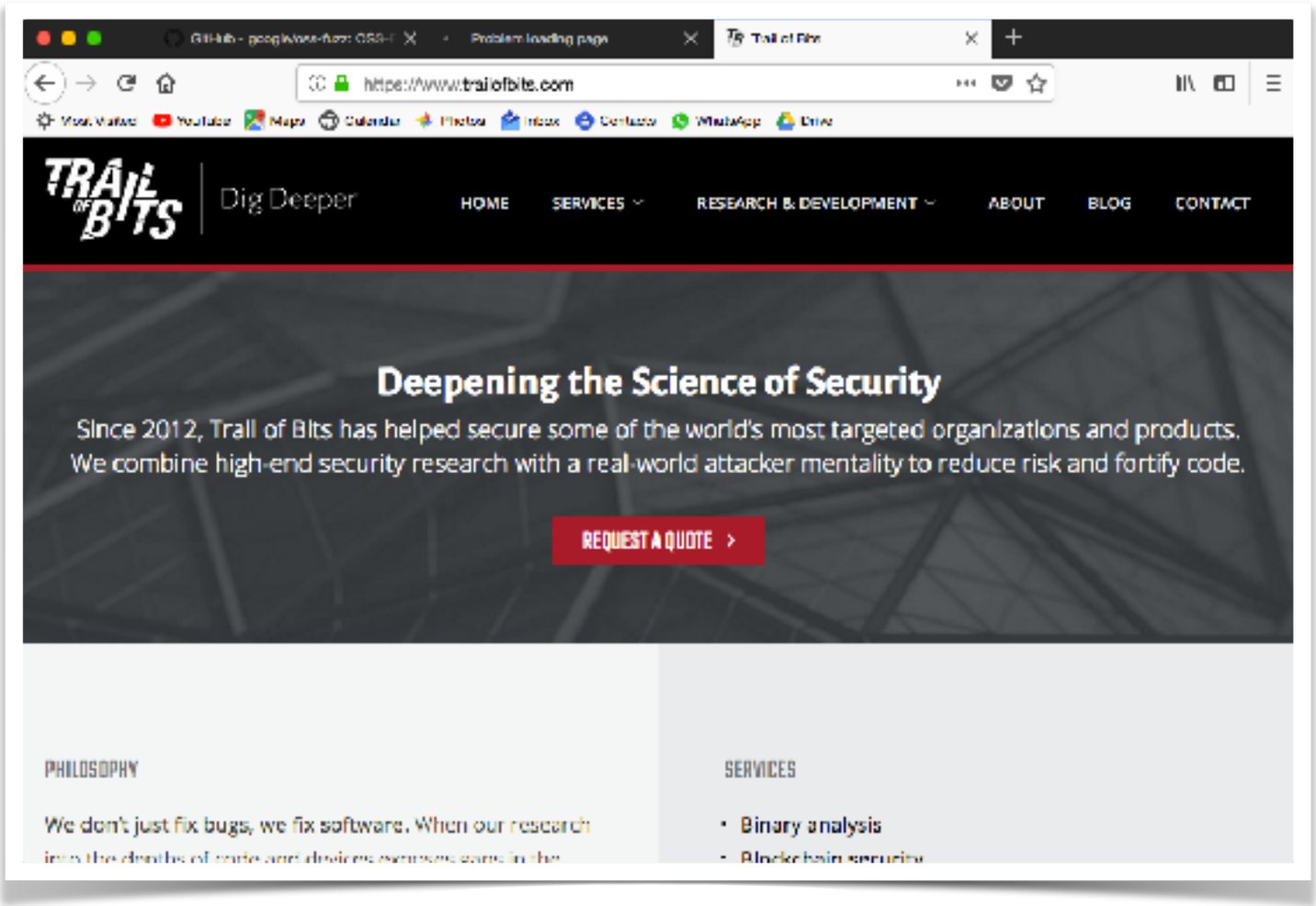
8 Used by

1 Discussion

3k Stars

176 Forks

- Fuzzing no se realiza únicamente en Google, Mozilla. También hay empresas especializadas en fuzzing (ejemplo: Trail of Bits)
- https://www.trailofbits.com
- https://twitter.com/trailofbits



Fuzz-Driven Development

- 2003: Test Driven Development (Kent Beck)
  - Insuficiente para seguridad
- 2017: Fuzz-Driven Development (Kostya Serebryany)
  - Cada API es un fuzz target
  - Tests == “Seed” para fuzzear
  - CI incluye Continuous Fuzzing

Recap

- Blackbox Fuzzing
- Greybox Fuzzing
- Boosted Greybox Fuzzing
- AFL / LibFuzzer (LLVM/C) + Sanitizers + asserts
- Grammar-based Fuzzing
- ClusterFuzz
- OSS-Fuzz