Fuzz Testing

Generación Automática de Casos de Test - 2018

+Andreas Zeller, Saarland University

Repaso: Random Testing

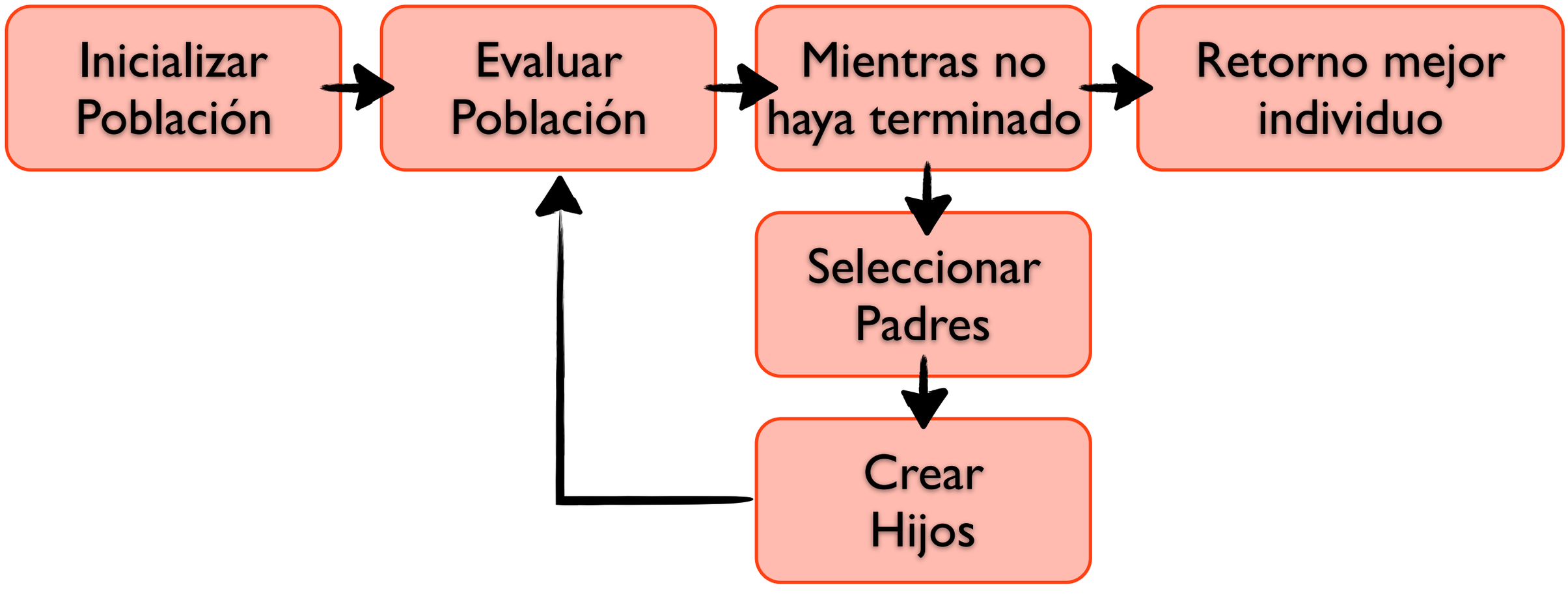
- Es una búsqueda (casi) completamente no guiada
- Si la técnica sistemática no es mejor que random testing, entonces no es valiosa
- Barata & fácil de implementar
- Funciona bastante bien en muchos casos

Repaso: Concolic Testing

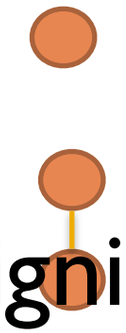
- Ejecuta concretamente el test pero guarda la path condition

- Utiliza un constraint solver para crear nuevos inputs

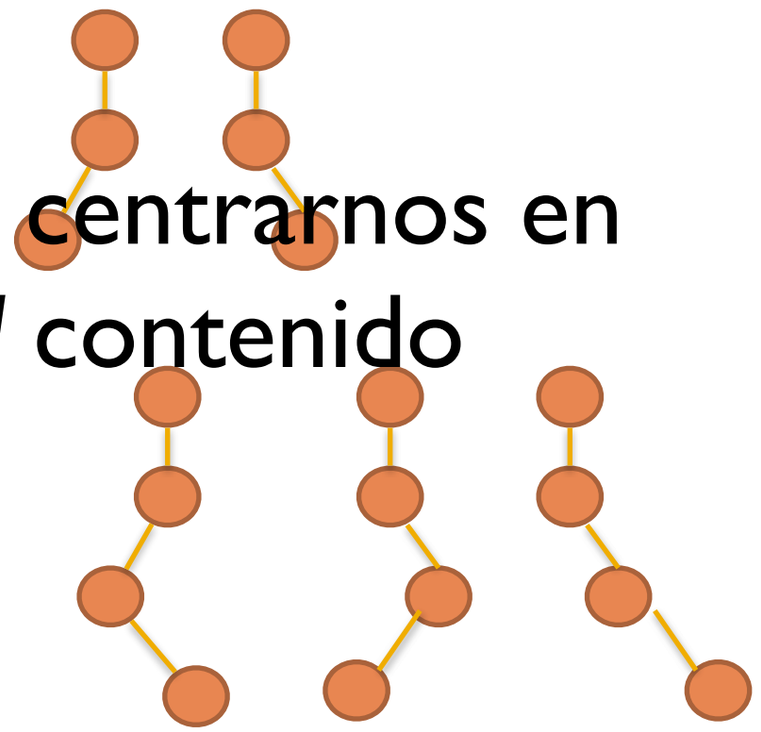
Repaso: Search-Based Testing



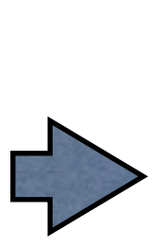
Repaso: Bounded Exhaustive Testing



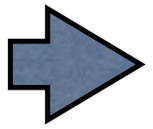
- Necesitamos identificar qué significa que dos inputs sean *equivalentes*
- Eso (generalmente) significa centrarnos en la e*structura en lugar de en el* contenido



Random Testing



Programa bajo Test



Oráculo

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

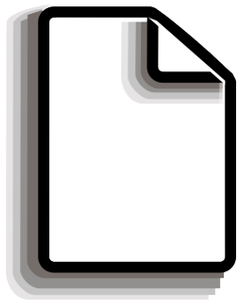
Fuzzing

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"

Inputs Típicos

- Formatos de archivos: documentos (DOCX,PDF), imágenes (PNG, ANI)



- Protocolos de red: Servidores HTTP, Clientes

- Interfaces de dominio: DOM

Fuzz Testing

“ab’d&gfdfggg”

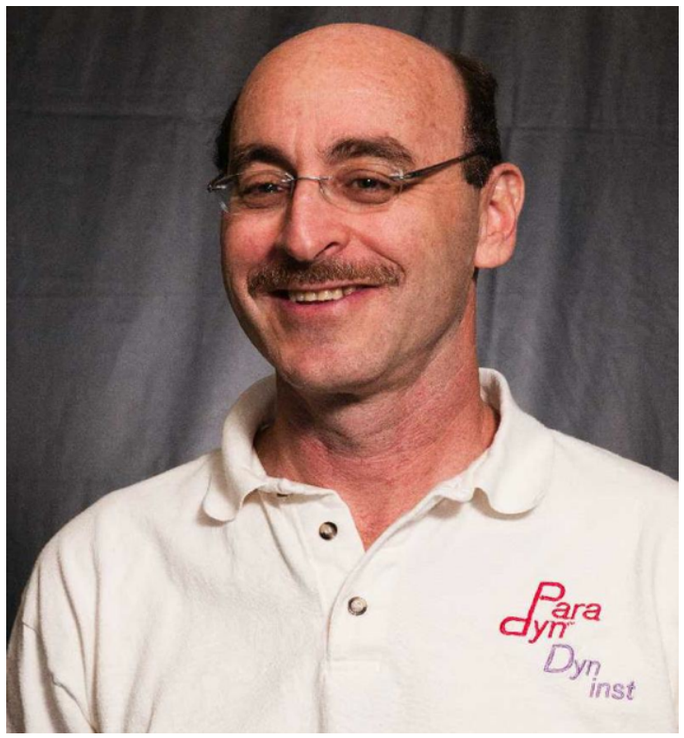
Black-Box Fuzzing

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)

fuzzer.py

```
import random

def fuzzer():
    # Strings up to 1024 characters long
    string_length = int(random.random() * 1024)

    # Fill it with ASCII 32..128 characters
    out = ""
    for i in range(0, string_length):
        out += chr(int(random.random() * 96 + 32))
    return out

if __name__ == "__main__":
    print fuzzer()
```

Fuzzer Output

[;x1-GPZ+wcckc];,N9J+?#6^6\e?]9lu2_%'4GX"0VUB[E/r
~fApu6b8<{%siq8Zh.6{V,hr?;{Ti.r3PIxMMMv6{xS^+'Hq!
AxB"YXRS@!Kd6;wtAMefFWM(`lJ_<1~o}z3K(CCzRH JIlvHz>_*.
\>JrlU32~eGP?IR=bF3+;y$3lodQ<B89!5"W2fK*vE7v{')KC-
i,c{<[~m!]o;{.'}Gj\(X}EtYetrpbY@aGZ1{P!AZU7x#4(Rtn!
q4nCwqol^y6}0lKo=*JK~;zMKV=9Nai:wxu{J&UV#HaU)*BiC<),`
+t*gka<W=Z.%T5WGHZpI30D<Pq>&]BS6R&j?#tP7iaV}-}`\?
[_[Z^LBMPG-FKj'\xwuZ1=Q`^`5,NQ@[!CuRzJ2DlvBy!
^zkhdf3C5PAkR?V hnl3='i2Qx]D
$qs4O`1@fevnG'2\11Vf3piU37@55ap\zIyl"'f,
$ee,J4Gw:cgNKLie3nx9(`efSlg6#[K"@WjhZ}r[Scun&sBCS,T[/
vY'pduwgzDlVNy7'rnzxNwI)(ynBa>%lb`;`9fG]P_0hdG~$@6
3]KAeEnQ7lU)3Pn,0)G/6N-wyzj/MTd#A;r

Demo

```
$ python fuzzer.py | prolog
$ python fuzzer.py | grep x
$ python fuzzer.py | tex
```

Fuzzing UNIX utilities

- Usar el output del fuzzer como un programa prolog:
  ```
  $ python fuzzer.py | prolog
  ```
- Usar el output del fuzzer cono la entrada a grep:
  ```
  $ python fuzzer.py | grep x
  ```
- Usar el output del fuzzer como documento TeX:
  ```
  $ python fuzzer.py | tex
  ```

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.”)

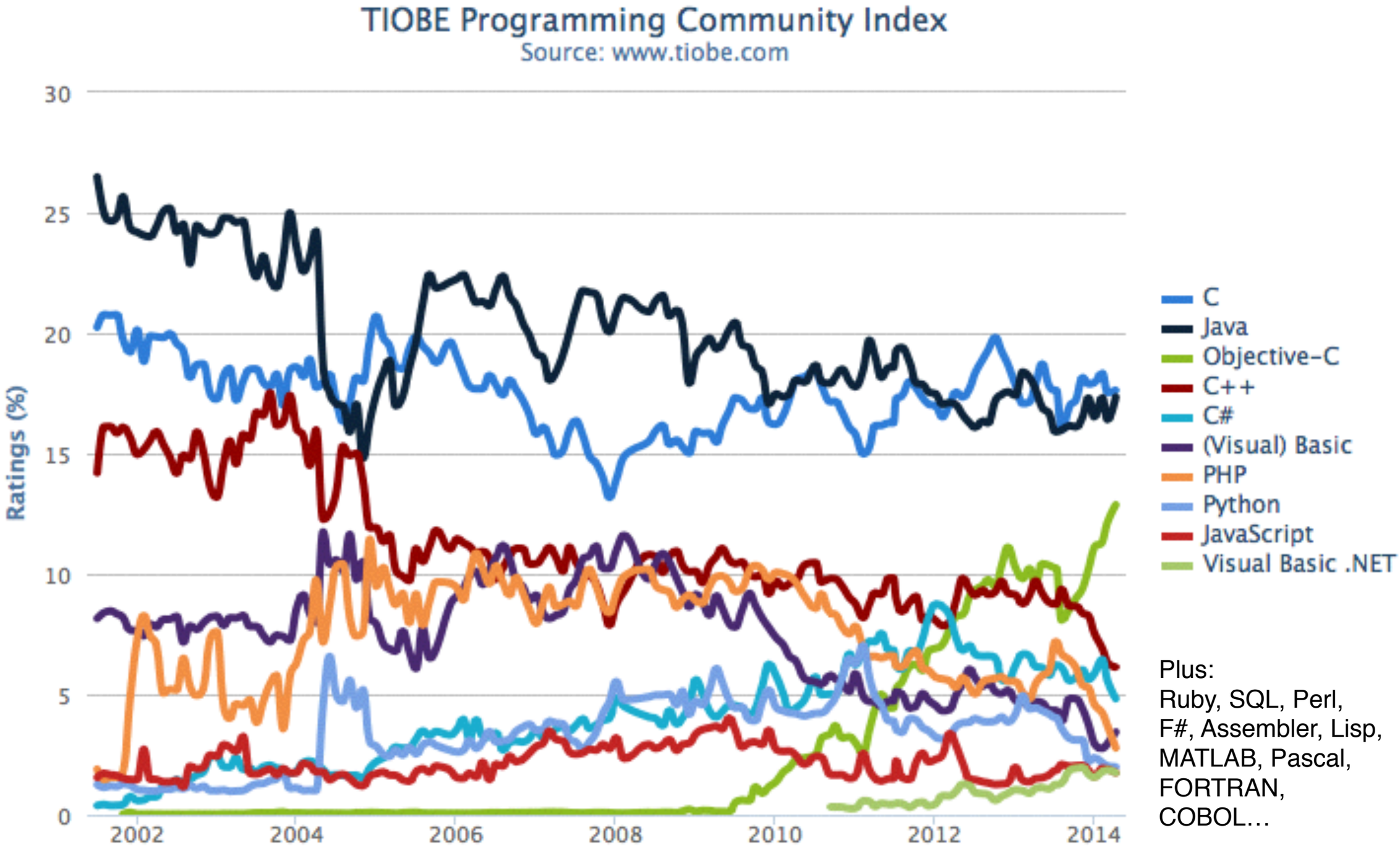
CRASH!

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



Plus:
Ruby, SQL, Perl,
F#, Assembler, Lisp,
MATLAB, Pascal,
FORTRAN,
COBOL…

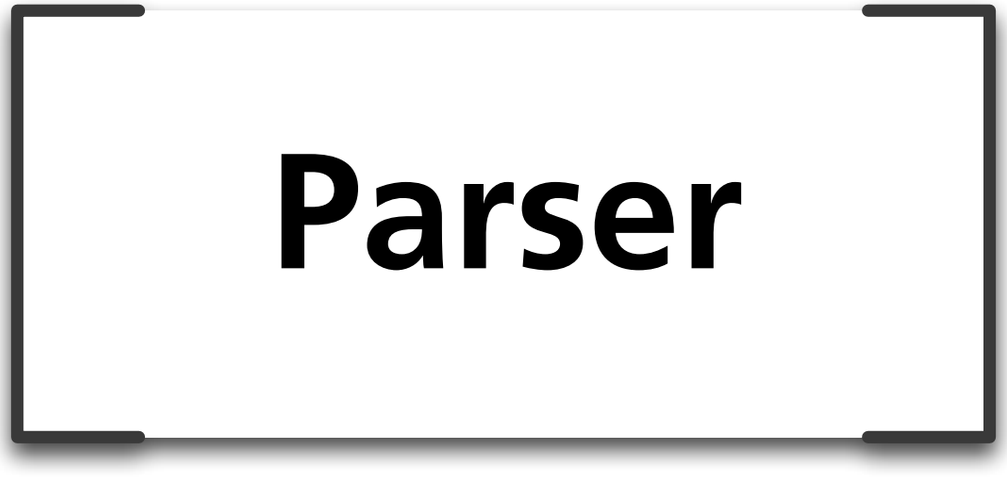
El Lenguaje C

Random Black-Box Fuzzing

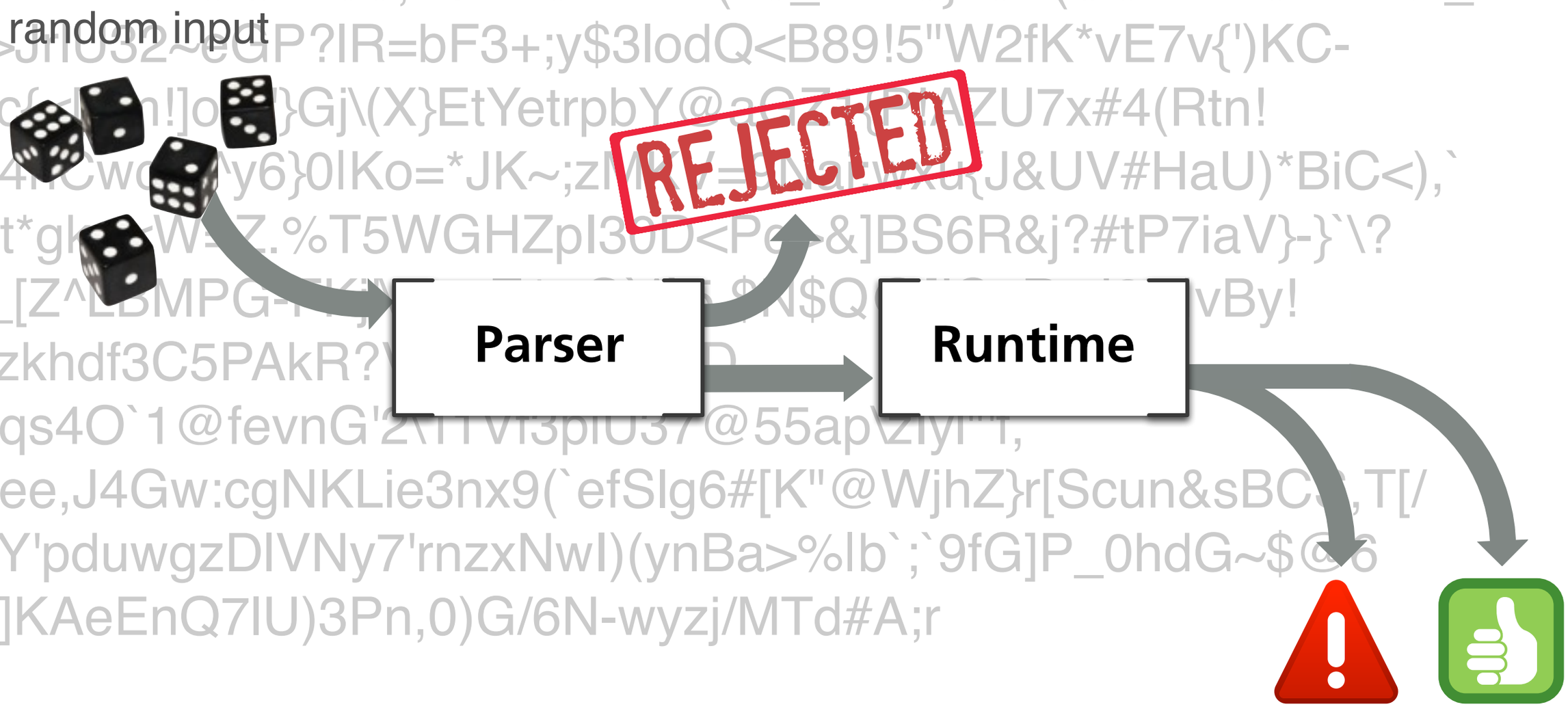
- Testing Black-box trata al programa bajo test como una "Caja Negra" (No se conoce nada interno del programa)
- Fuzzing, en su inicios, fue concebido como una técnica black-box
  - Random Testing a nivel de Sistema (System Level)
- Buenos resultados iniciales → crashes o hangs de entre 25-33% de los programas Unix (1989)

Nuevo Problema: Intérpretes

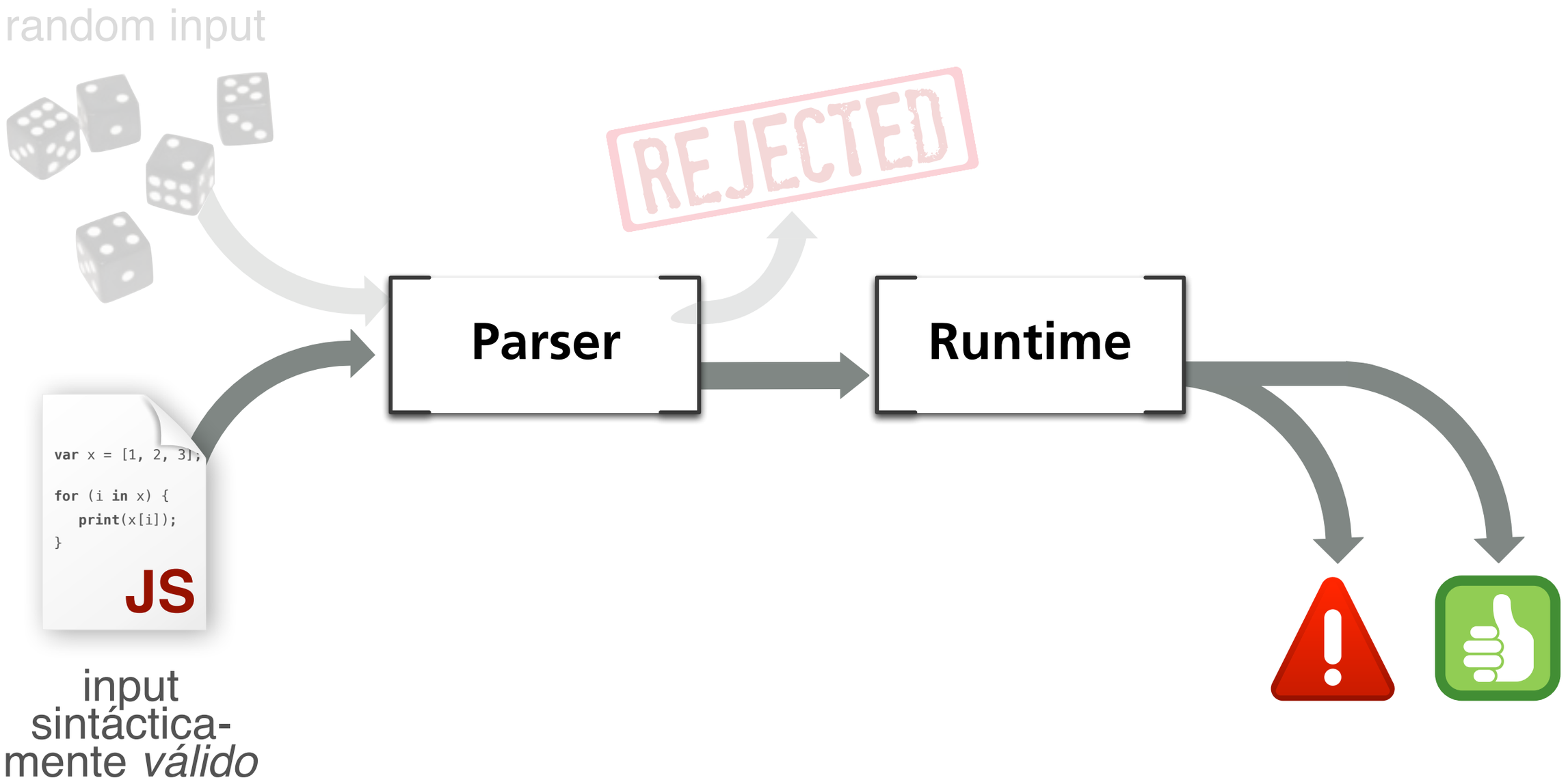
- Supongamos ahora que queremos testear el Intérprete de un Lenguaje de Programación –
- El Intérprete necesita compilar y ejecutar el programa



Random Inputs en Intérpretes



Fuzzing para Intérpretes



grammar-fuzz.py

- Queremos codificar una *gramática* para producir expresiones aritméticas como *strings*
- $START se expande a $EXPR, que puede expandirse a $TERM, $TERM + $TERM, etc.

```
grammar = [
    ("$START",  "$EXPR"),

    ("$EXPR",   "$EXPR + $TERM"),
    ("$EXPR",   "$EXPR – $TERM"),
    ("$EXPR",   "$TERM"),
```

grammar-fuzz.py

```
grammar = [
    ("$START",  "$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"),
```

```
    ("$FACTOR", "+$FACTOR"),
    ("$FACTOR", "-$FACTOR"),
    ("$FACTOR", "($EXPR)"),
    ("$FACTOR", "$INTEGER"),
    ("$FACTOR", "$INTEGER.$INTEGER"),

    ("$INTEGER", "$INTEGER$DIGIT"),
    ("$INTEGER", "$DIGIT"),

    ("$DIGIT", "1"),
    ("$DIGIT", "2"),
    ("$DIGIT", "3"),
    ("$DIGIT", "4"),
    ("$DIGIT", "5"),
    ("$DIGIT", "6"),
    ("$DIGIT", "7"),
    ("$DIGIT", "8"),
    ("$DIGIT", "9"),
    ("$DIGIT", "0")
]
```

```
    ("$FACTOR", "$INTEGER$INTEGER"),

    ("$INTEGER", "$INTEGER$DIGIT"),
    ("$INTEGER", "$DIGIT"),

    ("$DIGIT", "1"),
    ("$DIGIT", "2"),
    ("$DIGIT", "3"),
    ("$DIGIT", "4"),
    ("$DIGIT", "5"),
    ("$DIGIT", "6"),
    ("$DIGIT", "7"),
    ("$DIGIT", "8"),
    ("$DIGIT", "9"),
    ("$DIGIT", "0")
]

def apply(term, rule):
    (old, new) = rule
    # We replace the first occurrence;
    # this could also be some random occurrence
    return term.replace(old, new, 1)
```

```
def apply(term, rule):
    (old, new) = rule
    # We replace the first occurrence;
    # this could also be some random occurrence
    return term.replace(old, new, 1)

MAXSYMBOLS = 5

def produce():
    term = "$START"
    while term.count('$') > 0:
        # All rules have the same chance;
        # this could also be weighted
        index = random.randint(0, len(grammar) - 1)
        new_term = apply(term, grammar[index])
        if (new_term != term and
            new_term.count('$') < MAXSYMBOLS):
            print new_term
            term = new_term
    return term
```

$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

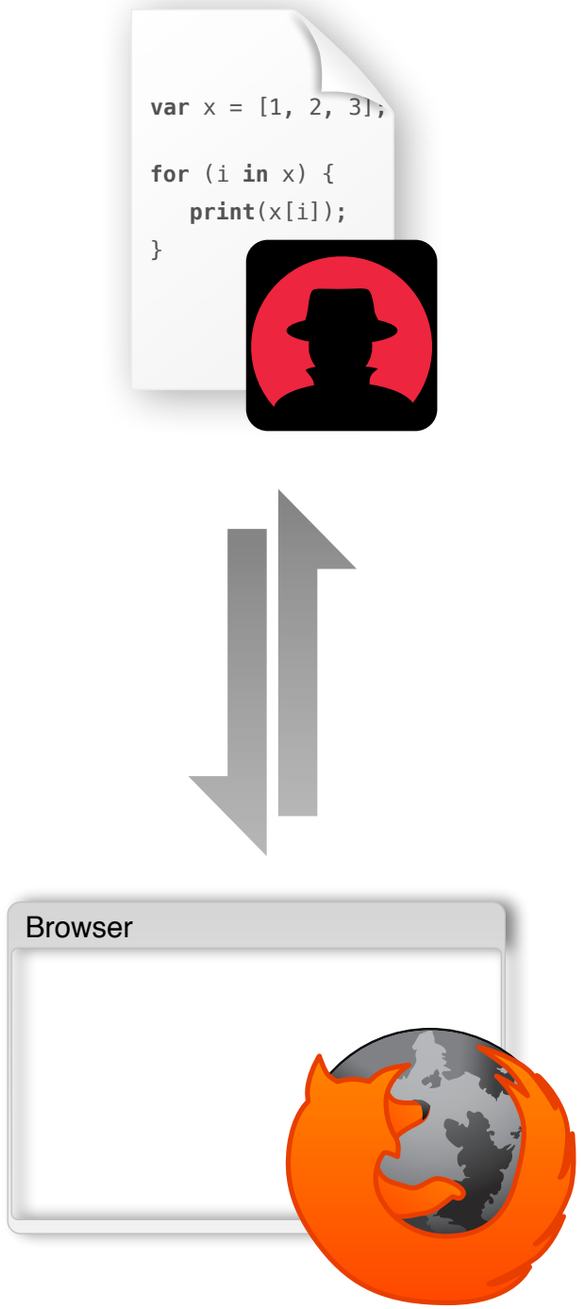
Demo

```
$ python grammar-fuzz.py
```

Fuzzing de Gramáticas

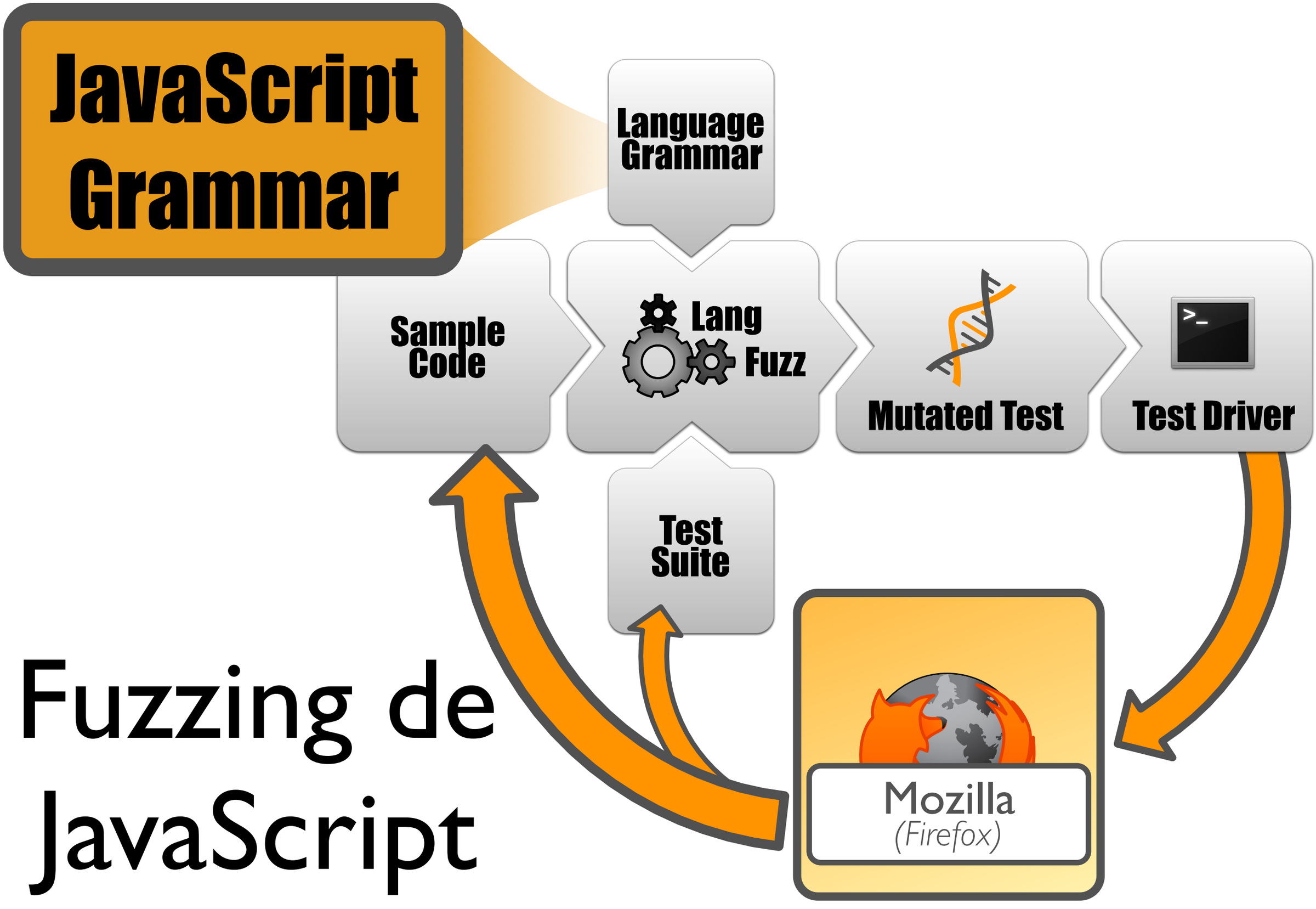
- LangFuzz es un Fuzz Tester que usa una *gramática* para generar inputs
- *Utiliza Inputs Válidos* para crear nuevos inputs

Gramática: JavaScript



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

Fuzzing de JavaScript



Gramática JavaScript

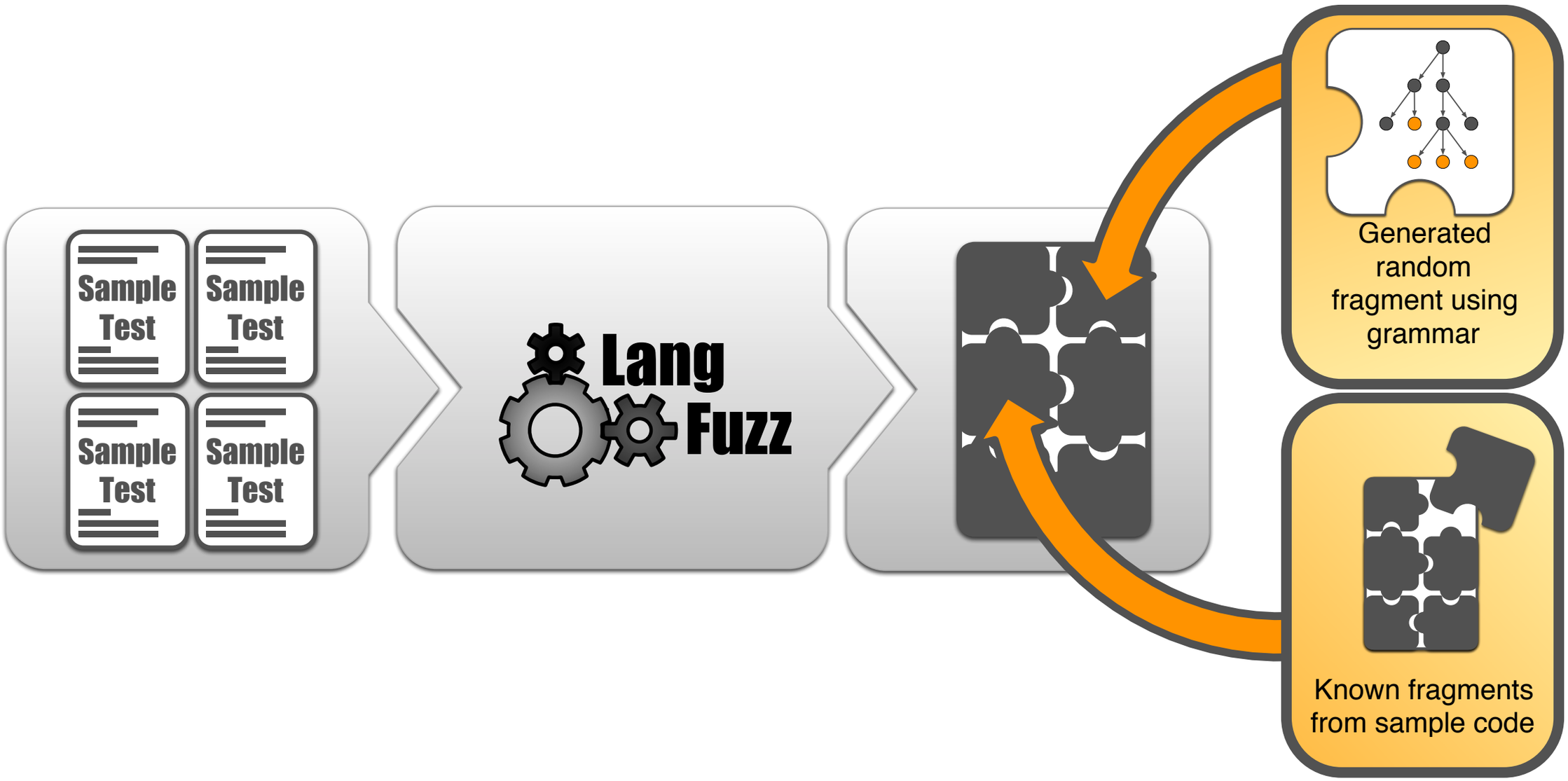
If Statement

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 ⇒

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 (Julio 2016)

En uso en Mozilla desde 2011,

2397 bugs en el engine JS.

En Google ClusterFuzz desde

2014, **~170 bugs** 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

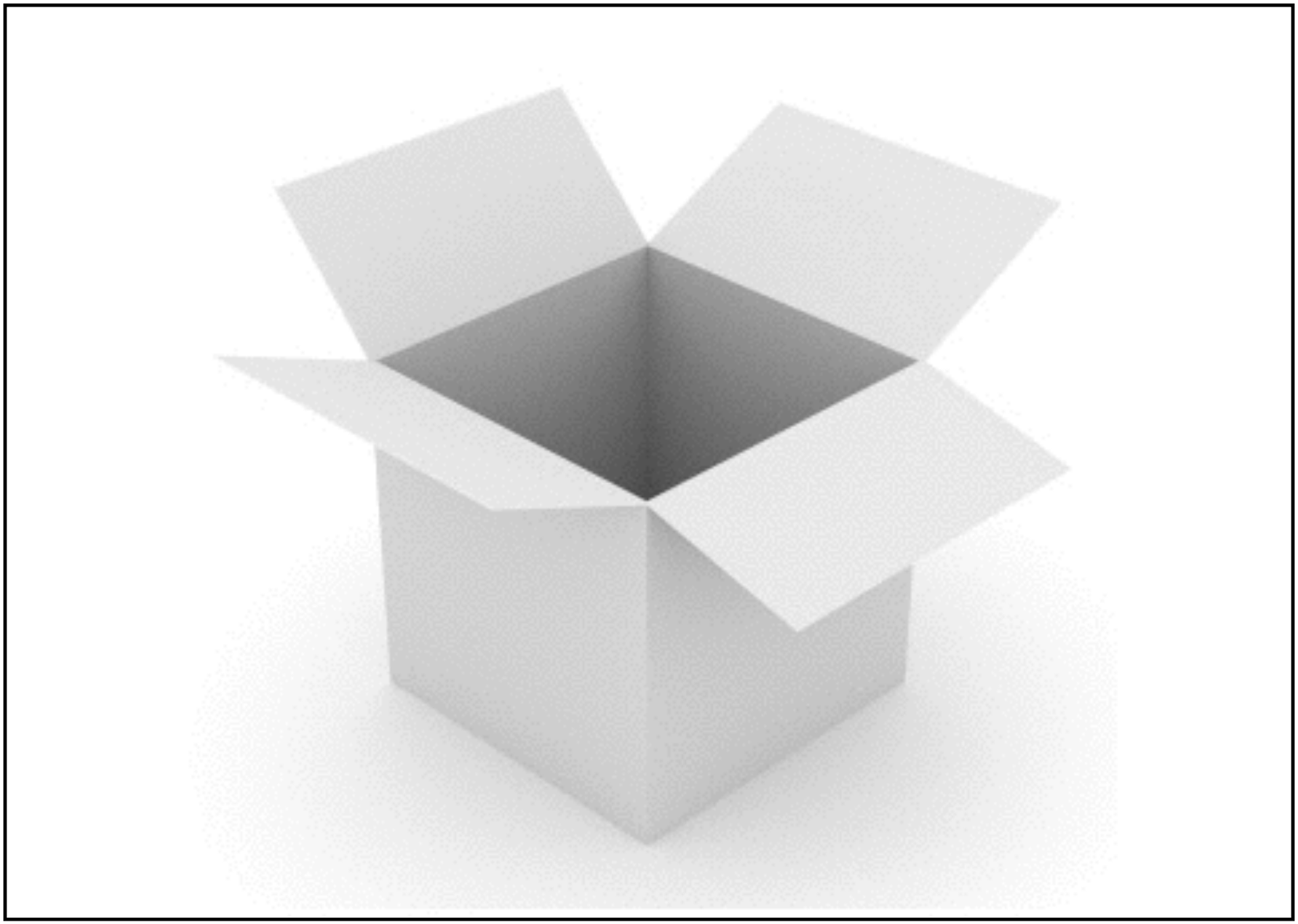
Mozilla Web and Services Hall Of Fame

Security Blog

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

White-box Fuzzing



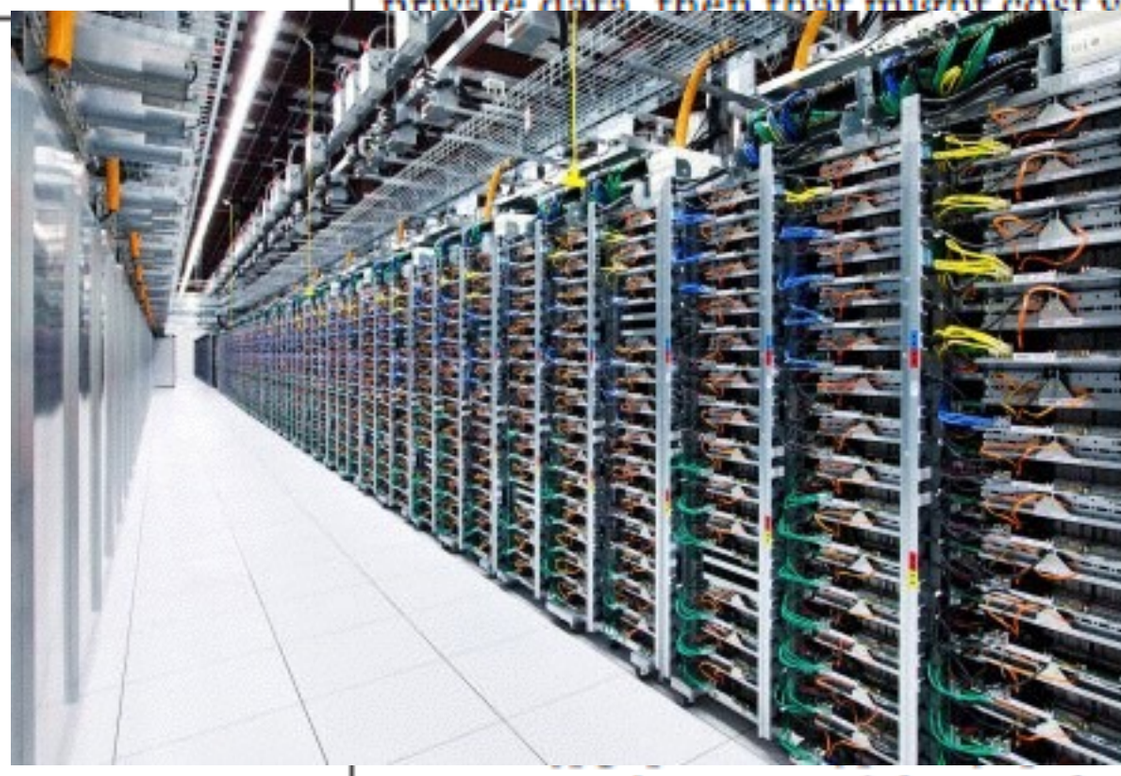
DOI:10.1145/2093548.2093564

Article development led by acmqueue
queue.acm.org

SAGE has had a remarkable impact at Microsoft.

BY PATRICE GODEFROID, MICHAEL Y. LEVIN, AND DAVID MOLNAR

SAGE: Whitebox Fuzzing for Security Testing



and its users millions of dollars. I
monthly security update costs y
$0.001 (one tenth of one cent) in ju
electricity or loss of productivity, th
this number multiplied by one b
lion people is $1 million. Of course
malware were spreading on your m
chine, possibly leaking some of yo
private data, then that might cost y

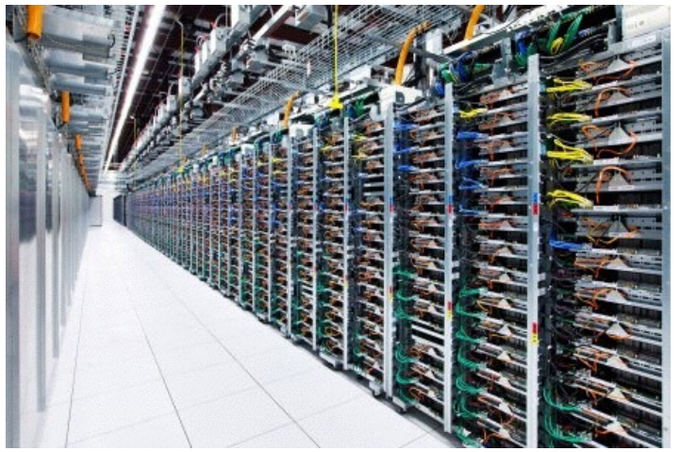
your operating system) has read t
image data, decoded it, created n
data structures with the decoded da
and passed those to the graphics ca
in your computer. If the code imp
menting that jpg parser contains
bug such as a buffer overflow that c
be triggered by a corrupted jpg imag
then the execution of this jpg pars
on your computer could potentially

Scalable Automated Guided Execution

- Primer herramienta en realizar **concolic execution** en binario x86
  - *"what you fuzz is what you ship,"* ya que los compiladores pueden realizar transformaciones que afectan la seguridad.
- Security bugs: los programadores pueden fallar en alocar memoria o manipular buffers apropiadamente, resultando en **vulnerabilidades**.

White-Box Fuzzing con SAGE

- La exploración de program paths se realiza paralelamente
- Cuanto más diverso y rico el conjunto inicial de inputs (ie seeds o semillas), mayor profundidad se logrará durante el fuzzing
- Utiliza heurísticas para maximizar el cubrimiento de código tan rápido como sea posible, con el objeto de encontrar bugs más rápidamente.
- Las **optimizaciones** son cruciales para manejar el **ENORME** tamaño de las path conditions.



SAGE

Microsoft

- *"Since 2008, SAGE has been running 24/7 on approximately 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs"*
- *"…has saved Microsoft millions of dollars as well as saved world time and energy, by avoiding expensive security patches to more than one billion PCs."*
- *"The software running on your PC has been affected by SAGE."*

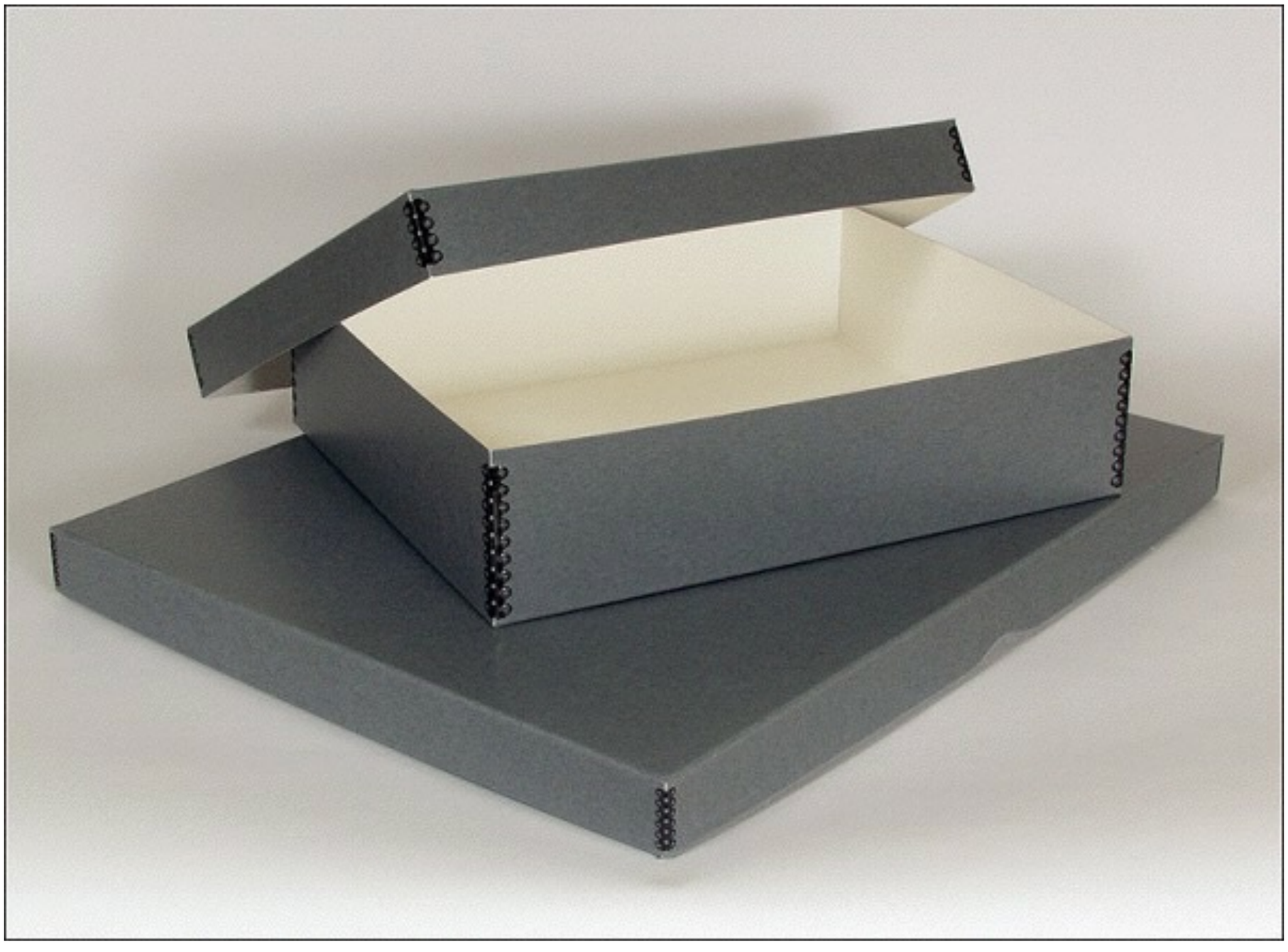
Microsoft Security Risk Detection

Sign up now for the Windows or Linux free trial

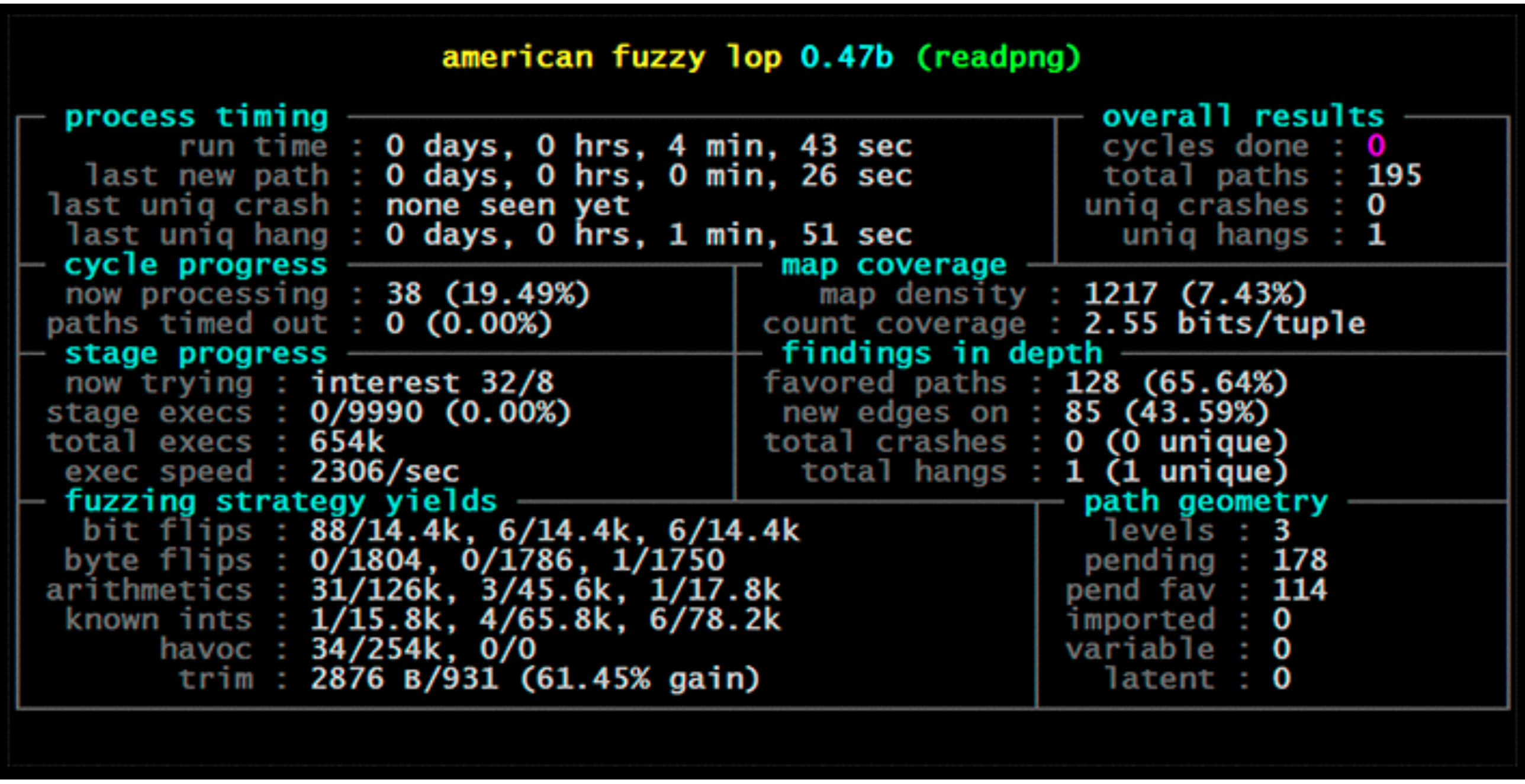
SIGN UP >


Microsoft Azure

Grey-box Fuzzing



American Fuzzy Lop



American Fuzzy Lop

The AFL approach

1) Encolar **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,

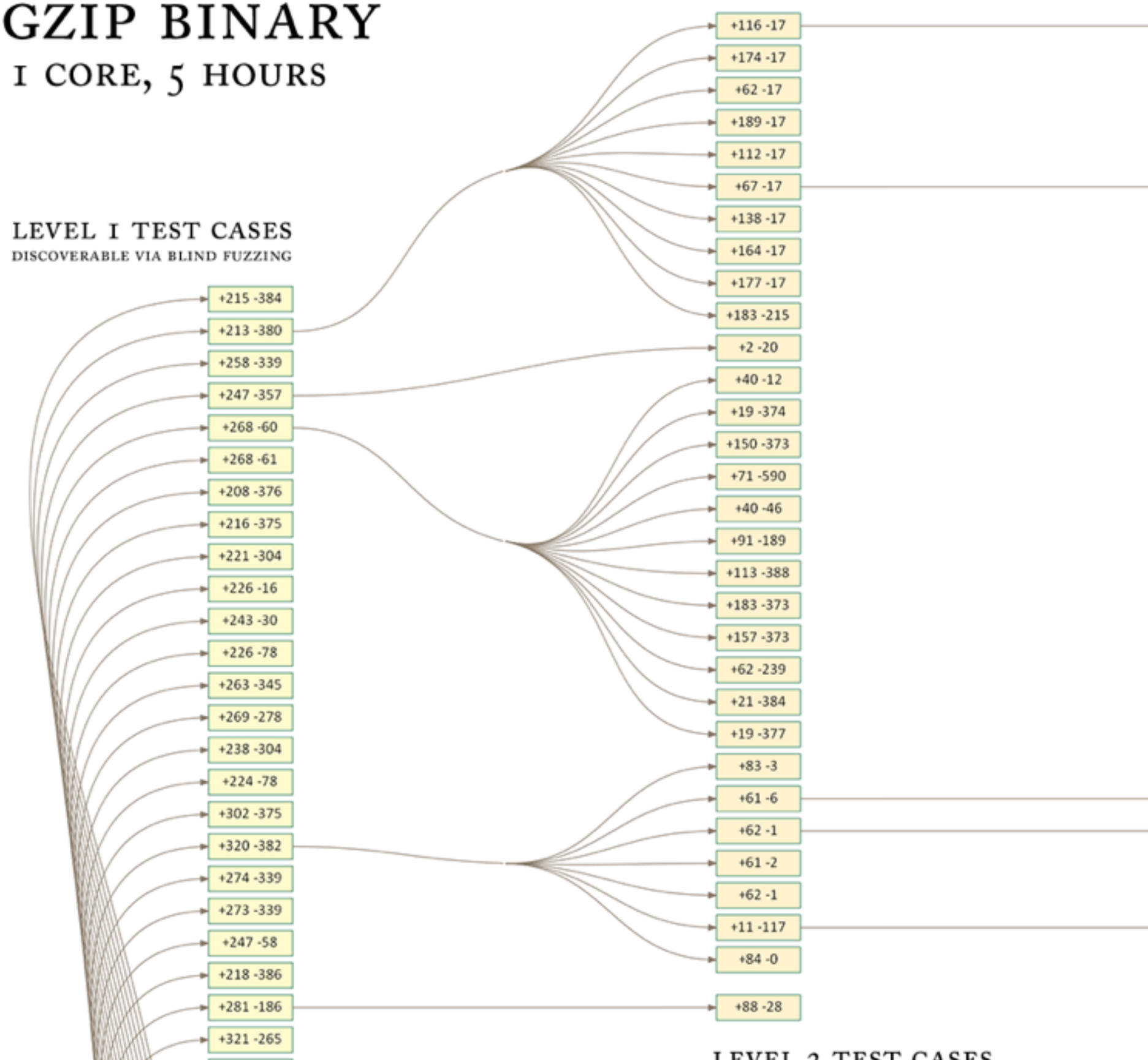
4) Aplicar un conjunto de mutaciones (cambios) al input utilizando una variedad de **estrategias tradicionales de fuzzing**,

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

6) Volver a 2).

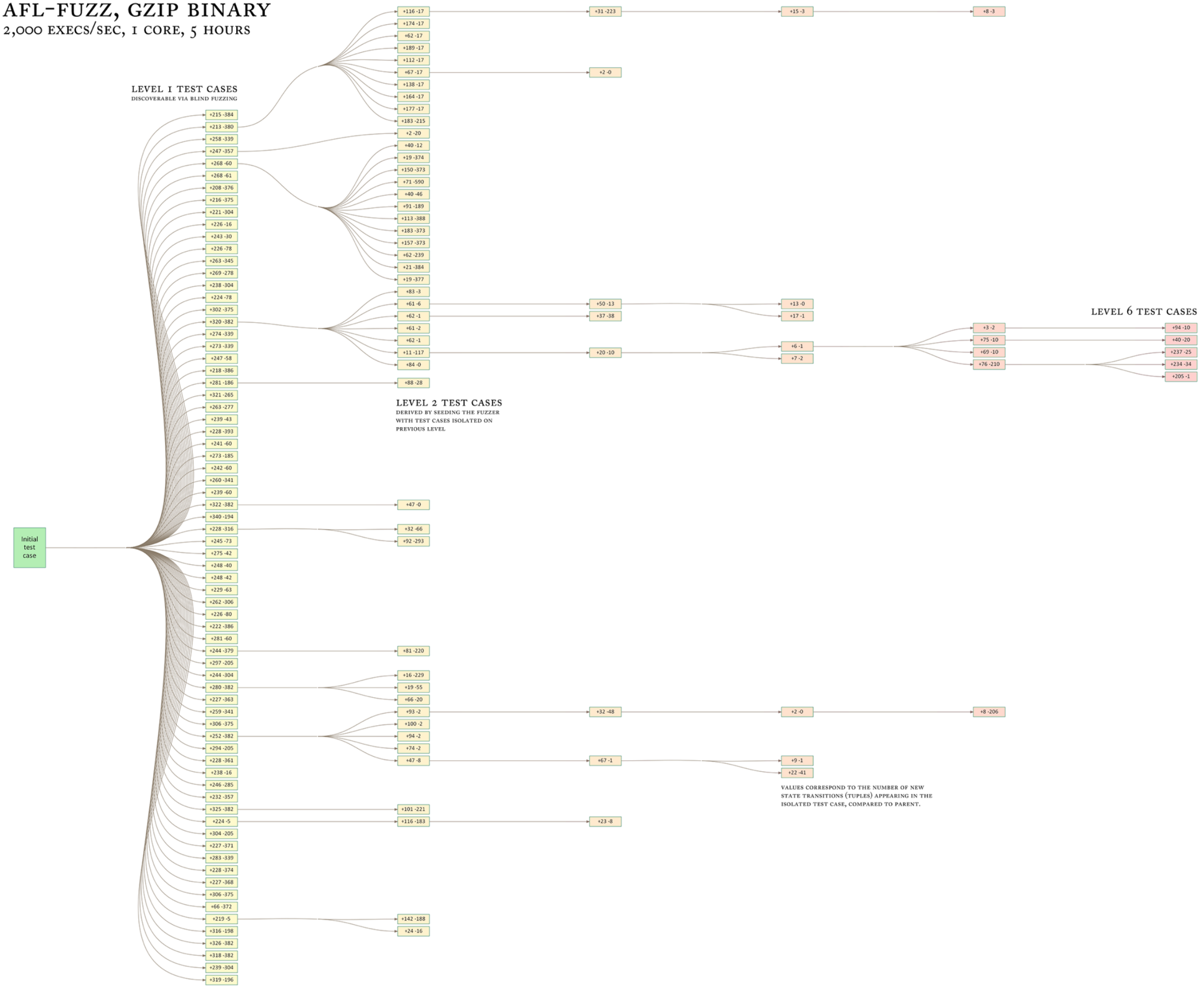
AFL-FUZZ, GZIP BINARY

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

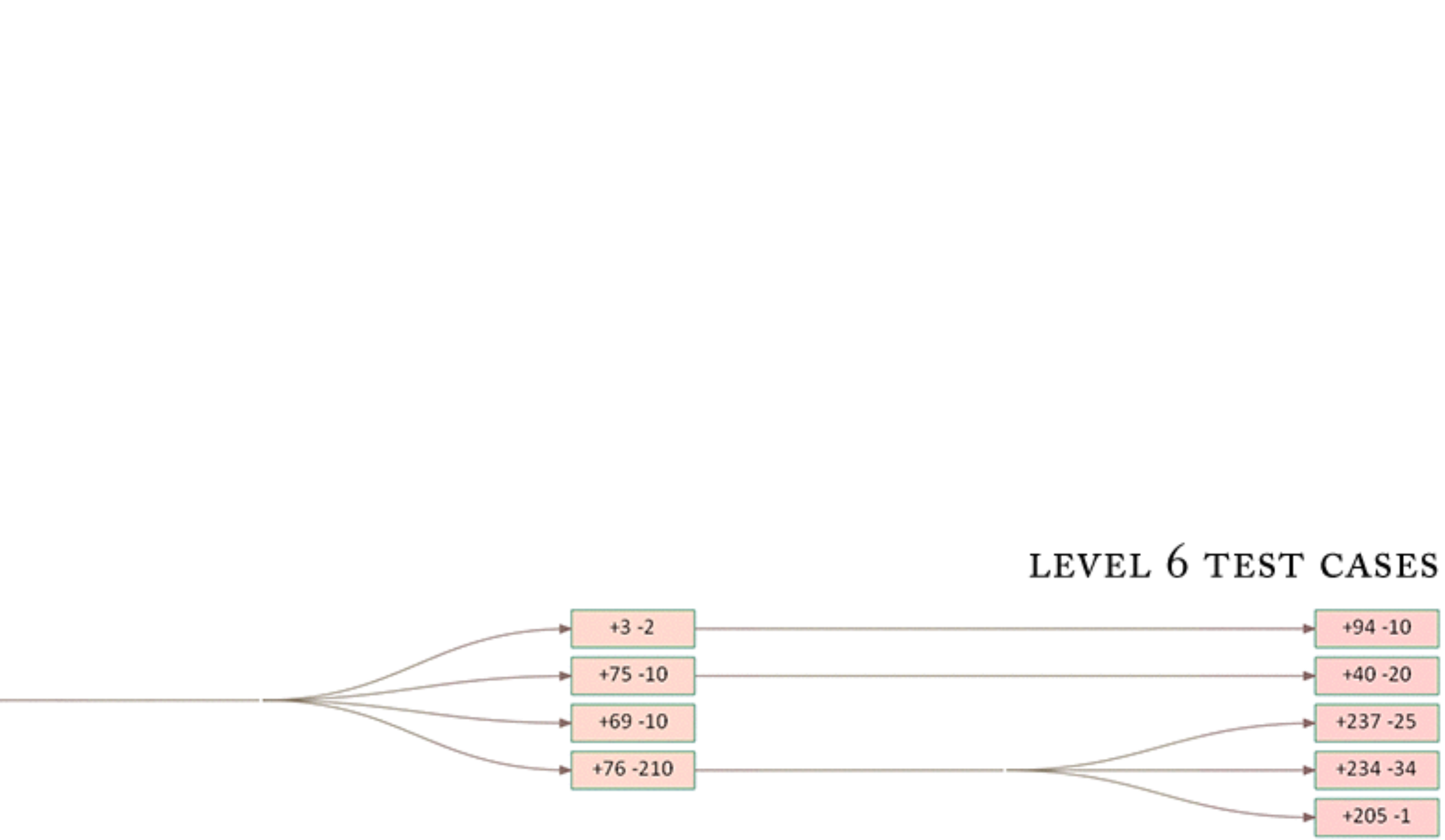


AFL-FUZZ, GZIP BINARY

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



LEVEL 6 TEST CASES



AFL Strategies

- **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

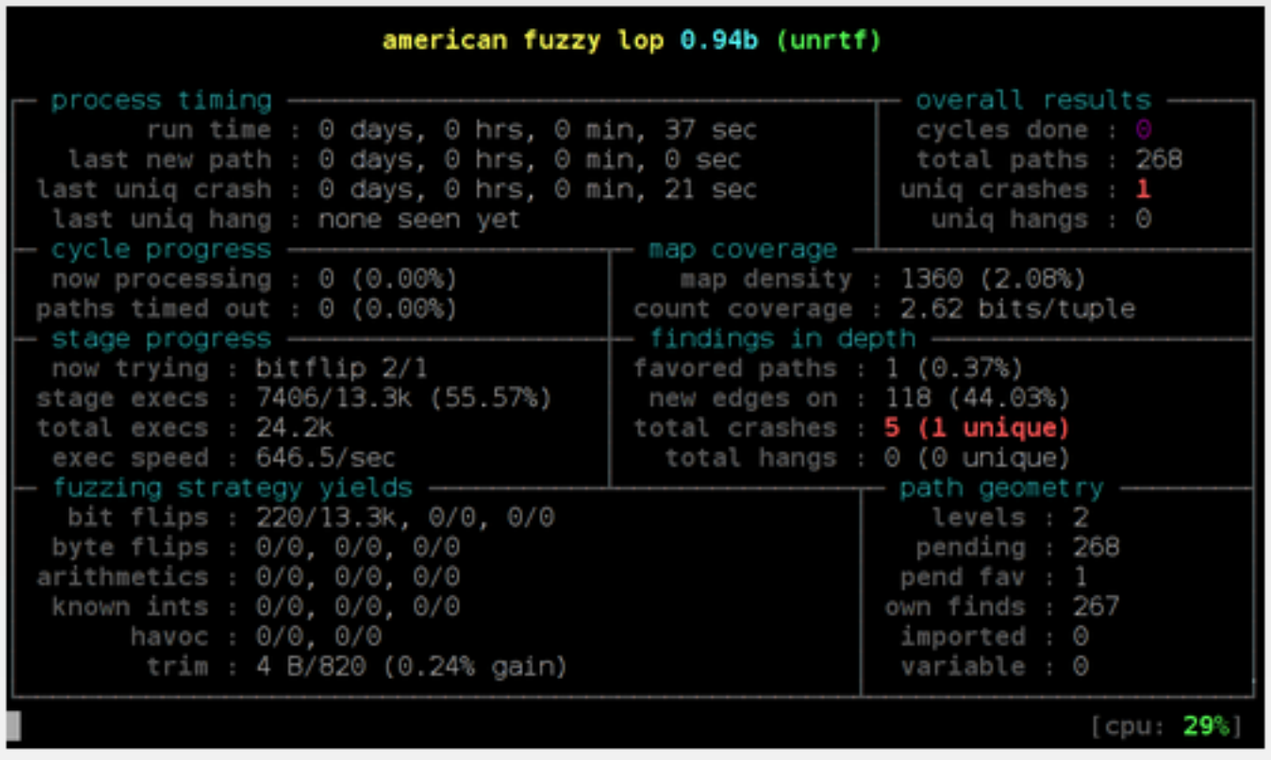
AFL Strategies

- https://lcamtuf.blogspot.com.ar/2014/08/binary-fuzzing-strategies-what-works.html

fuzzing-project.org

The Fuzzing Project

Fuzzing is a powerful strategy to find bugs in software. The idea is quite simple: Generate a large number of randomly malformed inputs for a software to parse and see what happens. If the program crashes then something is likely wrong. While fuzzing is a well-known strategy, it is surprisingly easy to find bugs, often with security implications, in widely used software.



Memory access errors are the errors most likely to be exposed are when fuzzing software that is written in C/C++. While they differ in the details (stack overflow, heap overflow, use after free, ...), the core problem is often the same: A software reads or writes to wrong memory locations.

A modern Linux or BSD system ships a large number of basic tools that do some kind of file displaying and parsing. In their current state most of these tools are not suitable for untrusted inputs.

On the other hand we have powerful tools these days that allow us to find and analyze these bugs, notably the fuzzing tool american fuzzy lop and the Address Sanitizer feature of gcc and clang.

The Fuzzing Project is trying to improve the state of things. I maintain **a list of software packages** with a rough categorization of how well they resist fuzzing and some helpful pointers how you can join the effort to improve the state of security in free software.

Más Allá de los Crashes

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

AddressSanitizer (C)

- 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

Recap: Fuzzing

- System Level
- Los Inputs Iniciales (seeds o semillas) son **IMPORTANTES**
- Apunta a Fallas de Seguridad
  - Crashes/Assertions
  - Latent Faults
- El Programa bajo Test es **UNSAFE**

Recap: Fuzzing

- Black-box Fuzzing == Random / Grammar-Based Testing
- White-box Fuzzing == Concolic Testing
- Grey-box Fuzzing == Search-Based Testing


FUZZ ALL THINGS
memegenerator.net

libFuzzer – a library for coverage-guided fuzz testing.

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

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

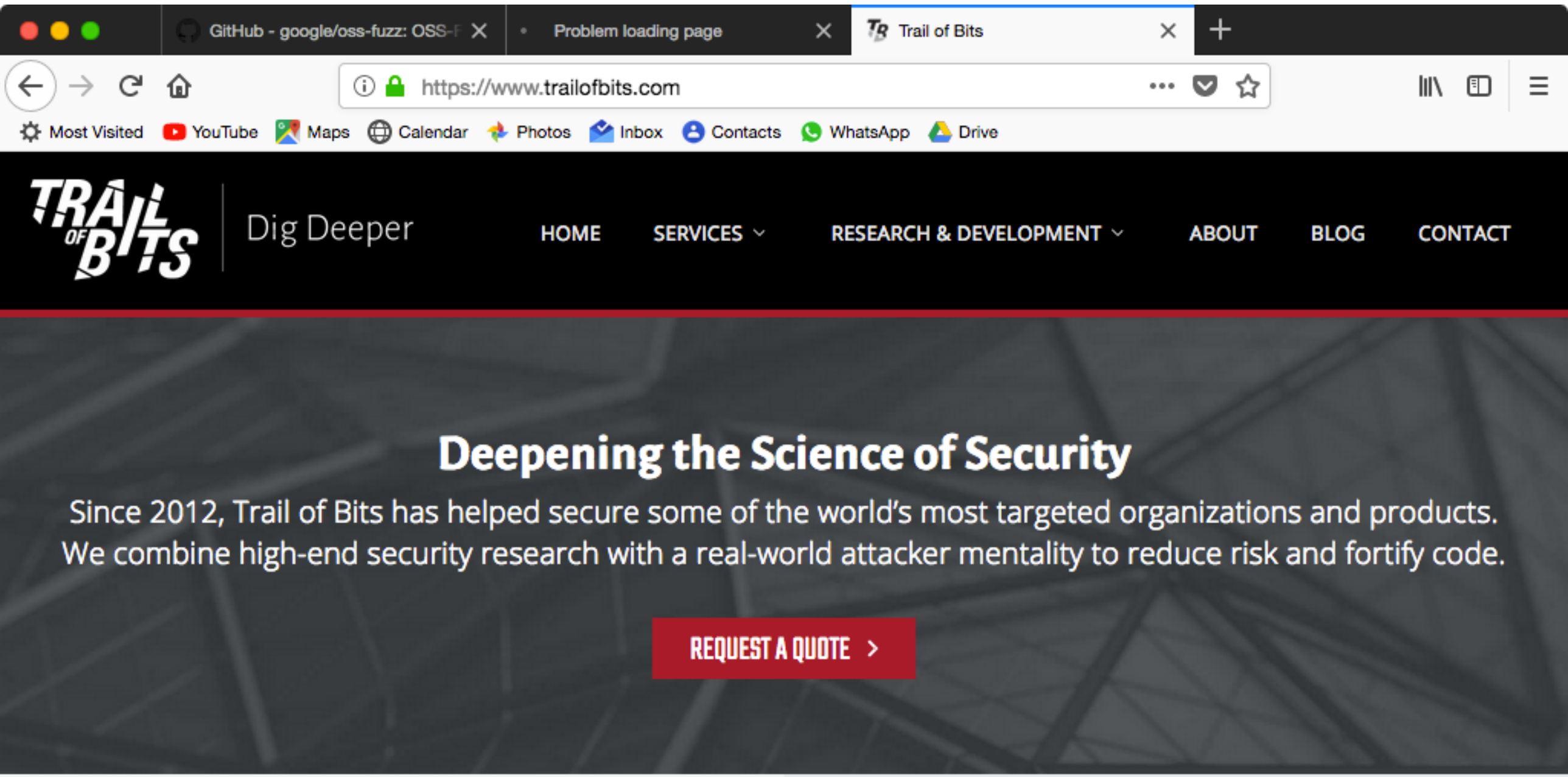
Fuzz testing is a well-known technique for uncovering various kinds of programming errors in software. Many of these detectable errors (e.g. buffer overflow) can have serious security implications.

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).



PHILOSOPHY

We don't just fix bugs, we fix software. When our research into the depths of code and devices exposes gaps in the

SERVICES

- Binary analysis
- Blockchain security

Simplificando el Input

[;x1-GPZ+wcckc];,N9J+?#6^6\e?]9lu2_%'4GX"0VUB[E/r
~fApu6b8<{%siq8Zh.6{V,hr?;{Ti.r3PIxMMMv6{xS^+'Hq!
AxB"YXRS@!Kd6;wtAMefFWM(`lJ_<1~o}z3K(CCzRH JIlvHz>_*.
\>JrlU32~eGP?lB–bF3+;y$3lodQ<B89I5"W2fK*yE7y{')KC-
i,c{<[~m!]o;{.'}C n!
q4nCwqol^y6}( )*BiC<),`
+t*gka<W=Z.% V}-}`\?
[_[Z^LBMPG-F By!
^zkhdf3C5PAk
$qs4O`1@fevnG 2\TTVI3plO37@55apZIyF`1,
$ee,J4Gw:cgNKLie3nx9(`efSlg6#[K"@WjhZ}r[Scun&sBCS,T[/
vY'pduwgzDlVNy7'rnzxNwI)(ynBa>%lb`;`9fG]P_0hdG~$@6
3]KAeEnQ7lU)3Pn,0)G/6N-wyzj/MTd#A;r

¿Qué parte del input es responsable de la falla?

Simplificando el Input

[;x1-GPZ+wcckc];,N9J+?#6^6\e?]9lu2_%'4GX"0VUB[E/r
~fApu6b8<{%siq8Zh.6{V,hr?;{Ti.r3PIxMMMv6{xS^+'Hq!
AxB"YXRS@!Kd6;wtAMefFWM(`lJ_<1~o}z3K(CCzRH JIlvHz>_*.
\>JrlU32~eGP?IR=bF3+;y$3lodQ<B89!5"W2fK*vE7v{')KC-
i,c{<[~m!]o;{.'}Gj\(X}EtYetrpbY@aGZ1{P!AZU7x#4(Rtn!
q4nCwqol^y6}0lKo=*JK~;zMKV=9Nai:wxu{J&UV#HaU)*BiC<),`
+t*gka<W=Z.%T5WGHZpI30D<Pq>&]BS6R&j?#tP7iaV}-}`\?
[_[Z^LBMPG-FKj'\xwuZ1=Q`^`5,NQ@[!CuRzJ2DlvBy!
^zkhdf3C5PAkR?V hnl3='i2Qx]D
$qs4O`1@fevnG'2\11Vf3piU37@55ap\zIyl"'f,
$ee,J4Gw:cgNKLie3nx9(`efSlg6#[K"@WjhZ}r[Scun&sBCS,T[/
vY'pduwgzDlVNy7'rnzxNwI)(ynBa>%lb`;`9fG]P_0hdG~$@6
3]KAeEnQ7lU)3Pn,0)G/6N-wyzj/MTd#A;r

Simplificando el Input

[;x1-GPZ+wcckc];,N9J+?#6^6\e?]9lu2_%'4GX"0VUB[E/r
~fApu6b8<{%siq8Zh.6{V,hr?;{Ti.r3PIxMMMv6{xS^+'Hq!
AxB"YXRS@!Kd6;wtAMefFWM(`lJ_<1~o}z3K(CCzRH JIlvHz>_*.
\>JrlU32~eGP?lR=bF3+;y$3lodQ<B89!5"W2fK*vE7v{')KC-
v{')
i,c{<[~m!]o;{.'}Gi\(X}EtYetrpbY@aGZ1{P!AZU7x#4(Rtn!
q4nCwqol^y6}0lKo=*JK~;zMKV=9Nai:wxu{J&UV#HaU)*BiC<),`
+t*gka<W=Z.%T5WGHZpl30D<Pq>&]BS6R&j?#tP7iaV}-}`\?
[_[Z^LBMPG-FKj'\xwuZ1=Q`^`5,NQ@[!CuRzJ2DlvBy!
^zkhdf3C5PAkR?V hnl3='i2Qx]D
$qs4O`1@fevnG'2\11Vf3piU37@55ap\zIyl"'f,
$ee,J4Gw:cgNKLie3nx9(`efSlg6#[K"@WjhZ}r[Scun&sBCS,T[/
vY'pduwgzDlVNy7'rnzxNwI)(ynBa>%lb`;`9fG]P_0hdG~$@6
3]KAeEnQ7lU)3Pn,0)G/6N-wyzj/MTd#A;r

¿Por qué simplificar?

- **Mejor comunicación:** Un test case simplificado es más fácil de entender.
- **Mejor debugging:** Test cases más chicos resultan en menos estados y trazas.
- **Identificar duplicados:** Un test simplificado subsume distintos duplicados.

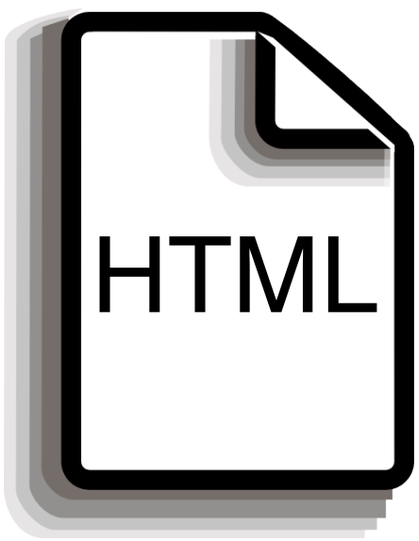
Experimentación

- Con *experimentación*, uno descubre si una circunstancia es relevante o no:
  - Omitimos la circunstancia y tratamos de reproducir el problema.
  - La circunstancia es relevante sii el problema no sigue ocurriendo.

Simplificando

- Por cada circunstancia del problema, chequear si es relevante para que el problema siga ocurriendo.
- Si no lo es, removarla del test case en cuestión.

Testing Firefox



Input

Crashes
Hangs
Assertions

Binary Search

- Procedemos por binary search. Descartar la mitad del input y chequear si el output es todavía erróneo.
- Si no lo es, restaurar el estado y descartar la otra mitad del input.

HTML input

¿Qué podemos hacer si *ambas mitades* pasan?

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

```
<SELECT NAME="priority" MULTIPLE SIZE=7>
```

Delta Debugging

- **Delta Debugging** toma un conjunto de *circunstancias* (un input) y un *oráculo.*
- A través de experimentación, computa el *subconjunto de circunstancias* necesarias para hacer que el input falle.
- Cada circunstancia retornada es *relevante* para la falla; en caso de removerla, la falla no ocurre.

ddmin en Acción

```
<SELECT NAME="priority" MULTIPLE SIZE=7> ✘
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✘
<SELECT NAME="priority" MULTIPLE SIZE=7> ✘
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✘
<SELECT NAME="p
<SELECT NAME="p
<SELECT NAME="p
<SELECT NAME="p
<SELECT NAME="priority" MULTIPLE SIZE=7> ✘
<SELECT NAME="priority" MULTIPLE SIZE=7> ✘
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✔
<SELECT NAME="priority" MULTIPLE SIZE=7> ✘
```

<select> alcanza para crashear

Delta Debugging

```
def test(s):
    global tests
    print tests, repr(s), len(s)
    tests += 1

    # Simulate the program under test
    if len(s) > 0 and '.' in s:
        return "FAIL"
    else:
        return "PASS"
```

```
def ddmin(s):
    # assert test("") == "PASS"
    # assert test(s) == "FAIL"

    n = 2     # Initial granularity
    while len(s) >= 2:
        start = 0
        subset_length = len(s) / n
        some_complement_is_failing = False

        while start < len(s):
            complement = (
                s[:start] + s[start + subset_length:])
            if test(complement) == "FAIL":
                s = complement
                n = max(n - 1, 2)
                some_complement_is_failing = True
                break

            start += subset_length

        if not some_complement_is_failing:
            if n == len(s):
                break
            n = min(n * 2, len(s))

    return s
```

Demo

```
$ python ddfuzz.py
```

ddmin and Fuzzing

[;x1-GPZ+wcckc];,N9J+?#6^6\e?]9lu2_%'4GX"0VUB[E/r
~fApu6b8<{%siq8Zh.6{V,hr?;{Ti.r3PIxMMMv6{xS^+'Hq!
AxB"YXRS@!Kd6;wtAMefFWM(`lJ_<1~o}z3K(CCzRH JIlvHz>_*.
\>JrlU32~eGP?IR=bF3+;y$3lodQ<B89!5"W2fK*vE7v{')KC-
i,c{<[~m!]o;{.'}Gj\(X}EtYetrpbY@aGZ1{P!AZU7x#4(Rtn!
q4nCwqol^y6}0lKo=*JK~;zMKV=9Nai:wxu{J&UV#HaU)*BiC<),`
+t*gka<W=Z.%T5WGHZpI30D<Pq>&]BS6R&j?#tP7iaV}-}`\?
[_[Z^LBMPG-FKj'\xwuZ1=Q`^`5,NQ@[!CuRzJ2DlvBy!
^zkhdf3C5PAkR?V hnl3='i2Qx]D
$qs4O`1@fevnG'2\11Vf3piU37@55ap\zIyl"'f,
$ee,J4Gw:cgNKLie3nx9(`efSlg6#[K"@WjhZ}r[Scun&sBCS,T[/
vY'pduwgzDlVNy7'rnzxNwI)(ynBa>%lb`;`9fG]P_0hdG~$@6
3]KAeEnQ7lU)3Pn,0)G/6N-wyzj/MTd#A;r

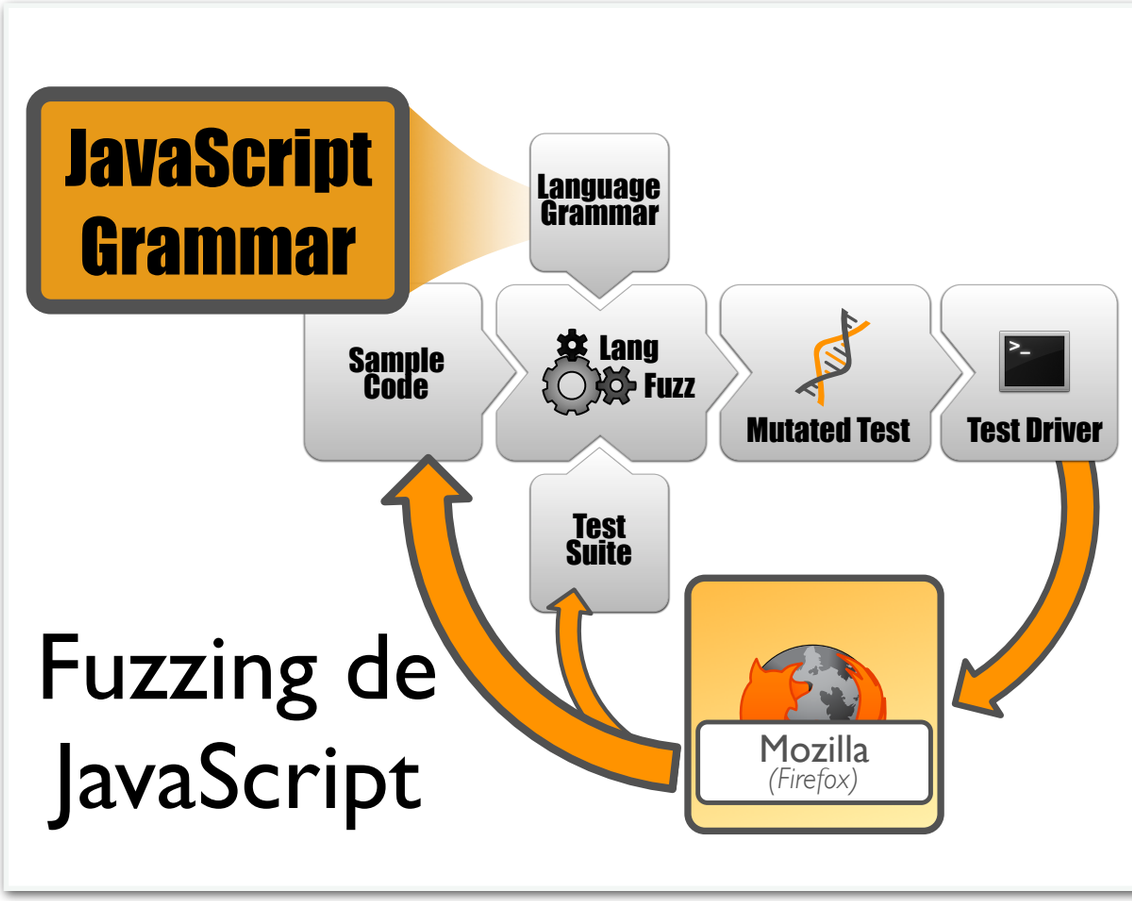
ddmin and Fuzzing

[;x1-GPZ+wcckc];,N9J+?#6^6\e?]9lu2_%'4GX"0VUB[E/r
~fApu6b8<{%siq8Zh.6{V,hr?;{Ti.r3PIxMMMv6{xS^+'Hq!
AxB"YXRS@!Kd6;wtAMefFWM(`lJ_<1~o}z3K(CCzRH JIlvHz>_*.
\>JrlU32~eGP?lR=bF3+;y$3lodQ<B89!5"W2fK*vE7v{')KC-
v{')
i,c{<[~m!]o;{.'}Gi\(X}EtYetrpbY@aGZ1{P!AZU7x#4(Rtn!
q4nCwqol^y6}0lKo=*JK~;zMKV=9Nai:wxu{J&UV#HaU)*BiC<),`
+t*gka<W=Z.%T5WGHZpl30D<Pq>&]BS6R&j?#tP7iaV}-}`\?
[_[Z^LBMPG-FKj'\xwuZ1=Q`^`5,NQ@[!CuRzJ2DlvBy!
^zkhdf3C5PAkR?V hnl3='i2Qx]D
$qs4O`1@fevnG'2\11Vf3piU37@55ap\zIyl"'f,
$ee,J4Gw:cgNKLie3nx9(`efSlg6#[K"@WjhZ}r[Scun&sBCS,T[/
vY'pduwgzDlVNy7'rnzxNwI)(ynBa>%lb`;`9fG]P_0hdG~$@6
3]KAeEnQ7lU)3Pn,0)G/6N-wyzj/MTd#A;r

Fuzzing

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

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



Fuzzing de JavaScript

DOI:10.1145/2093548.2093564

Article development led by acmqueue
queue.acm.org

SAGE has had a remarkable impact at Microsoft.

BY PATRICE GODEFROID, MICHAEL Y. LEVIN, AND DAVID MOLNAR

SAGE: Whitebox Fuzzing for Security Testing

and its users millions of dollars. I monthly security update costs y $0.001 (one tenth of one cent) in j electricity or loss of productivity, th this number multiplied by one l lion people is $1 million. Of course malware were spreading on your n chine, possibly leaking some of yc private data, then that might cost

your operating system) has read t image data, decoded it, created n data structures with the decoded da and passed those to the graphics ca in your computer. If the code imp menting that jpg parser contains bug such as a buffer overflow that c be triggered by a corrupted jpg ima then the execution of this jpg par on your computer could potentially

American Fuzzy Lop