

# Fuzzing

# Fuzzing

“Finding bugs by bombarding target with nonconform data”

- Think: Flip a few bits in a PDF, then start Acrobat with that PDF
- Just more automated

```
while True; do:  
    cp orig.pdf iteration.pdf  
    flip-random-bits.sh iteration.pdf  
    xpdf iteration.pdf  
done
```

# Fuzzer

A program which generates new “random” inputs, and feeds it to the target program.

Mutation-based:

- Modify existing test samples
- Shuffle, change, erase, insert

Grammar-based:

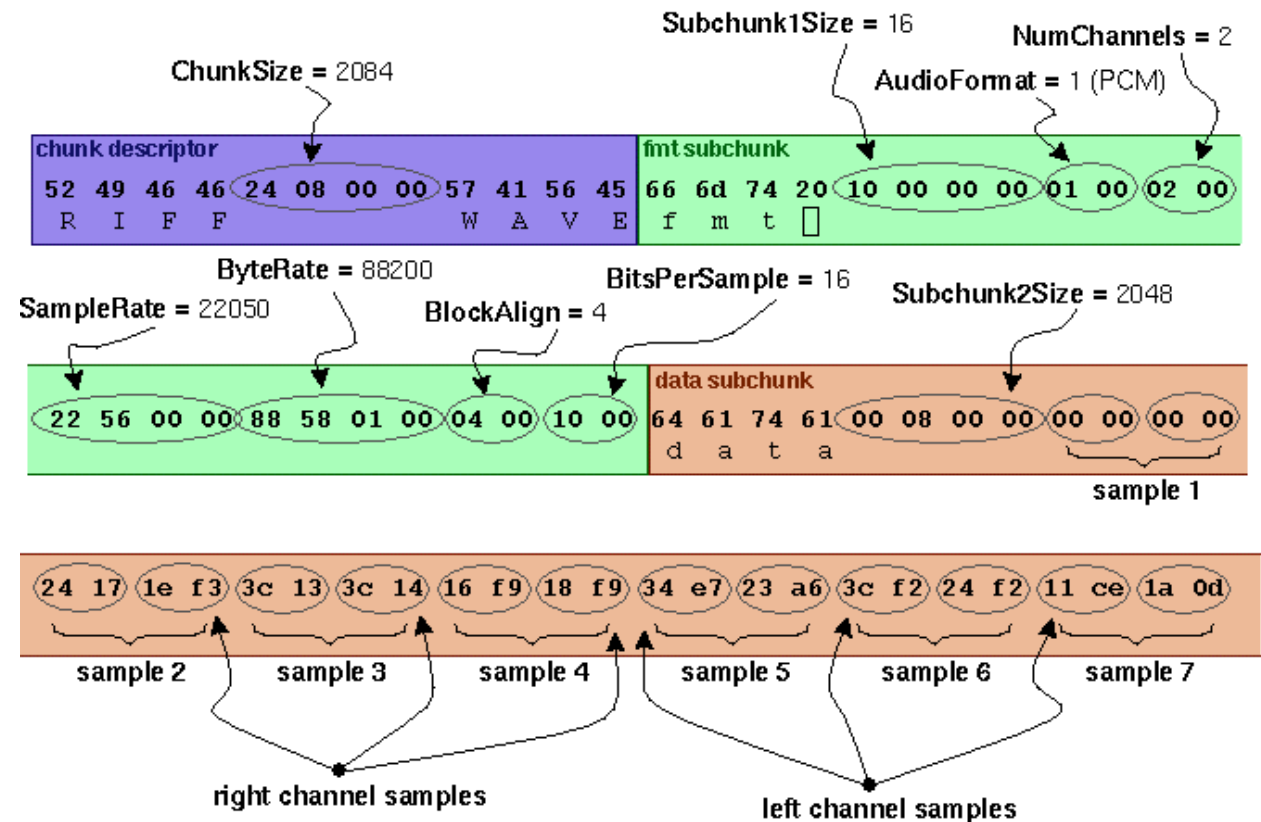
- Define new test sample based on models, templates, RFCs or documentation

# Fuzzer: Mutation-based

Mutation fuzzing examples:

- Ffmpeg: Movie files
- Winamp: MP3 files
- Antivirus: ELF files

Take an input file, modify it a bit, continue



# Fuzzer: Mutation-based

Steps:

- Create input corpus
- Select an input
- Modify input file (“fuzz it”)
- Start program with input file
- Identify crashes

# Fuzzer: Grammar-based

Grammar-based fuzzing:

- Browser: JavaScript
- Browser: HTML
- FTP, HTTP, ...

Cannot just bit flip etc, as it is not a binary protocol

```
alert(1);
```

- is valid:

```
alfrt(1);
```

- is garbage

# Fuzzer: Grammar-based

- Create a random output based on grammar
  - Start program with input file
  - Identify crashes
- 
- Well Known: Peach Fuzzer

# Fuzzer: Grammar-based

## Domato

```
!varformat fuzzvar%05d
!lineguard try { <line> } catch(e) {}

!begin lines
<new element> = document.getElementById("<string min=97 max=122>");
<element>.doSomething();
!end lines
```

If we instruct the engine to generate 5 lines, we may end up with something like:

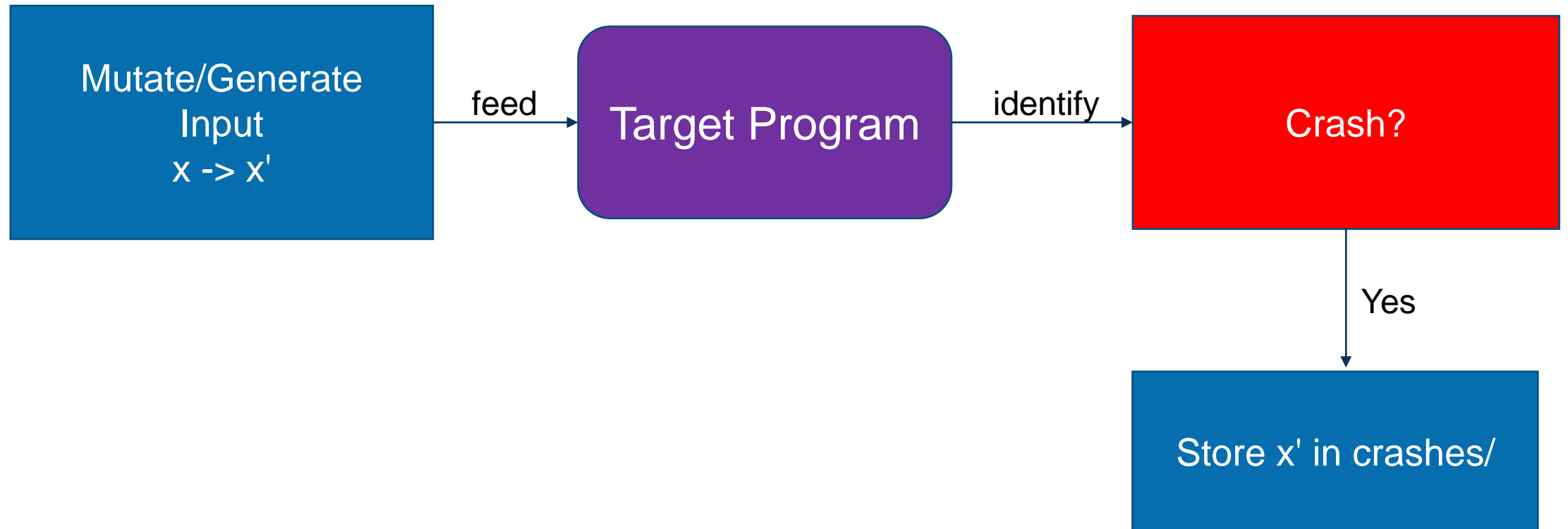
```
try { var00001 = document.getElementById("hw"); } catch(e) {}
try { var00001.doSomething(); } catch(e) {}
try { var00002 = document.getElementById("feezcqbnf"); } catch(e) {}
try { var00002.doSomething(); } catch(e) {}
try { var00001.doSomething(); } catch(e) {}
```



# Fuzzer: Grammar-based

```
HTTP-date      = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date  = weekday "," SP date2 SP time SP "GMT"
asctime-date  = wkday SP date3 SP time SP 4DIGIT
date1         = 2DIGIT SP month SP 4DIGIT
               ; day month year (e.g., 02 Jun 1982)
date2         = 2DIGIT "-" month "-" 2DIGIT
               ; day-month-year (e.g., 02-Jun-82)
date3         = month SP ( 2DIGIT | ( SP 1DIGIT ) )
               ; month day (e.g., Jun 2)
time          = 2DIGIT ":" 2DIGIT ":" 2DIGIT
               ; 00:00:00 - 23:59:59
wkday         = "Mon" | "Tue" | "Wed"
               | "Thu" | "Fri" | "Sat" | "Sun"
weekday       = "Monday" | "Tuesday" | "Wednesday"
               | "Thursday" | "Friday" | "Saturday" | "Sunday"
month         = "Jan" | "Feb" | "Mar" | "Apr"
               | "May" | "Jun" | "Jul" | "Aug"
               | "Sep" | "Oct" | "Nov" | "Dec"
```

# Traditional fuzzing - dumb, inefficient, brute force



**AFL**

## AFL

- Introduced “Code Coverage” to the masses
- Observe program to identify, IF new branches have been taken.
  - Save corresponding input file (which produces new code coverage)

# AFL

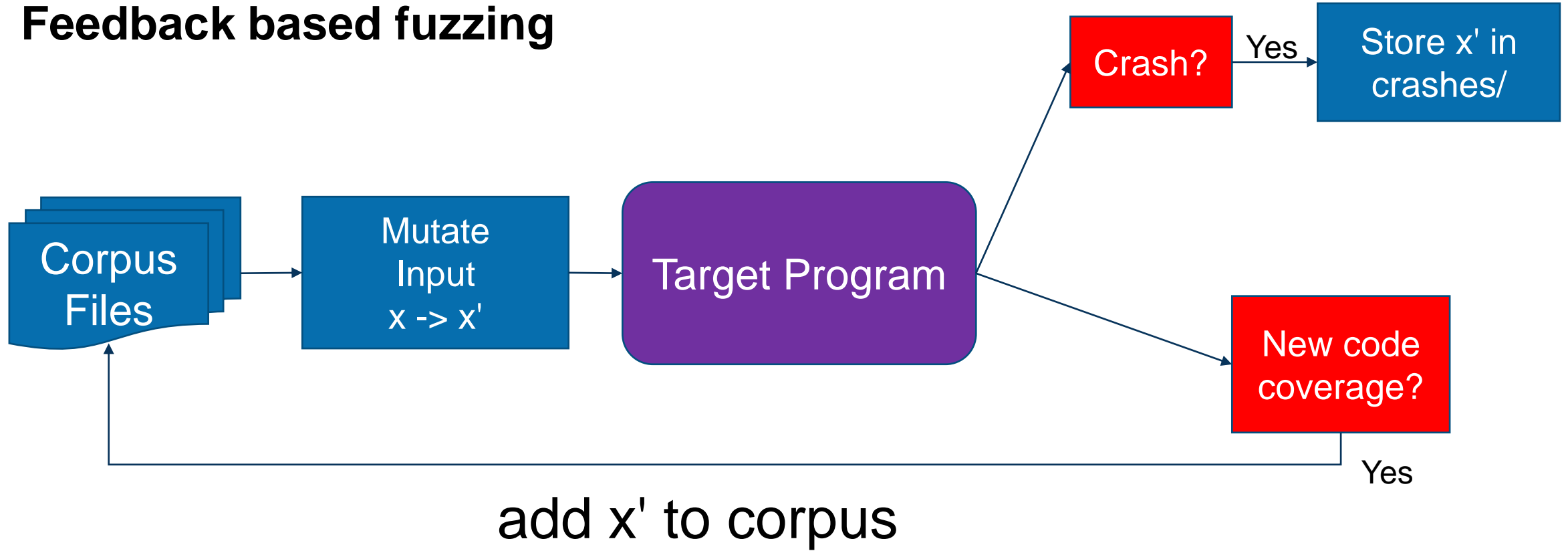
## american fuzzy lop (2.38b)

*American fuzzy lop* is a security-oriented [fuzzer](#) that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact [synthesized corpora](#) produced by the tool are also useful for seeding other, more labor- or resource-intensive testing regimes down the road.

american fuzzy lop 0.47b (readpng)			
<b>process timing</b>		<b>overall results</b>	
run time : 0 days, 0 hrs, 4 min, 43 sec		cycles done : 0	
last new path : 0 days, 0 hrs, 0 min, 26 sec		total paths : 195	
last uniq crash : none seen yet		uniq crashes : 0	
last uniq hang : 0 days, 0 hrs, 1 min, 51 sec		uniq hangs : 1	
<b>cycle progress</b>		<b>map coverage</b>	
now processing : 38 (19.49%)		map density : 1217 (7.43%)	
paths timed out : 0 (0.00%)		count coverage : 2.55 bits/tuple	
<b>stage progress</b>		<b>findings in depth</b>	
now trying : interest 32/8		favored paths : 128 (65.64%)	
stage execs : 0/9990 (0.00%)		new edges on : 85 (43.59%)	
total execs : 654k		total crashes : 0 (0 unique)	
exec speed : 2306/sec		total hangs : 1 (1 unique)	
<b>fuzzing strategy yields</b>		<b>path geometry</b>	
bit flips : 88/14.4k, 6/14.4k, 6/14.4k		levels : 3	
byte flips : 0/1804, 0/1786, 1/1750		pending : 178	
arithmetics : 31/126k, 3/45.6k, 1/17.8k		pend fav : 114	
known ints : 1/15.8k, 4/65.8k, 6/78.2k		imported : 0	
havoc : 34/254k, 0/0		variable : 0	
trim : 2876 B/931 (61.45% gain)		latent : 0	

Compared to other instrumented fuzzers, *afl-fuzz* is designed to be practical: it has modest performance overhead, uses a variety of highly effective fuzzing strategies and effort minimization tricks, requires [essentially no configuration](#), and seamlessly handles complex, real-world use cases - say, common image parsing or file compression libraries.

# Feedback based fuzzing



# Feedback based fuzzing

- "Observe" program to see if a new input (mutated from corpus) reaches new code path
  - This is being done by adding code in the compile process which tracks which functions get called in what order

# Fuzzer

Corpus:  
1. **X**



Function X

Function Y

Function 1

Function 2

Function 0

Function A

Function B

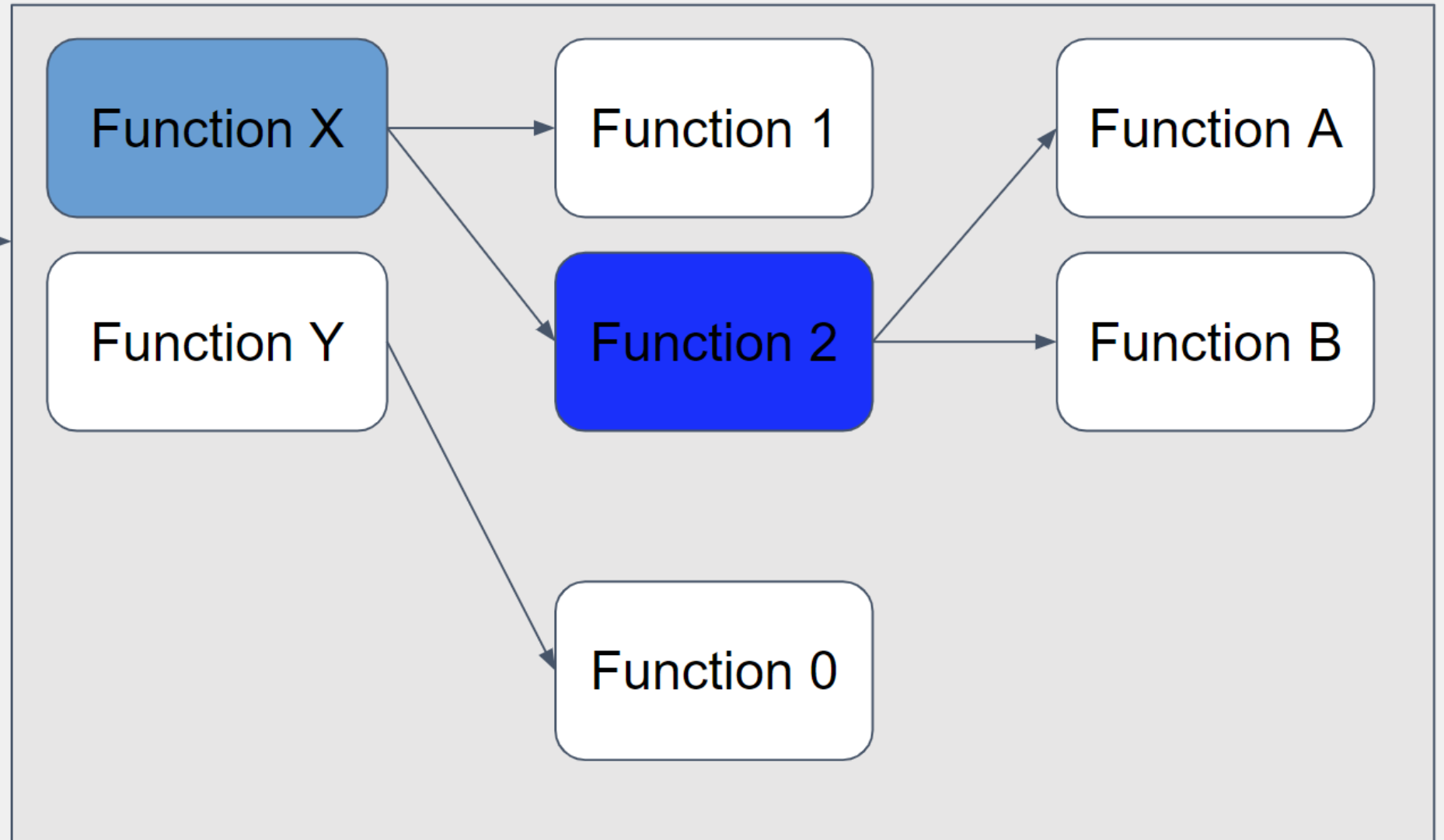




# Fuzzer

Corpus:

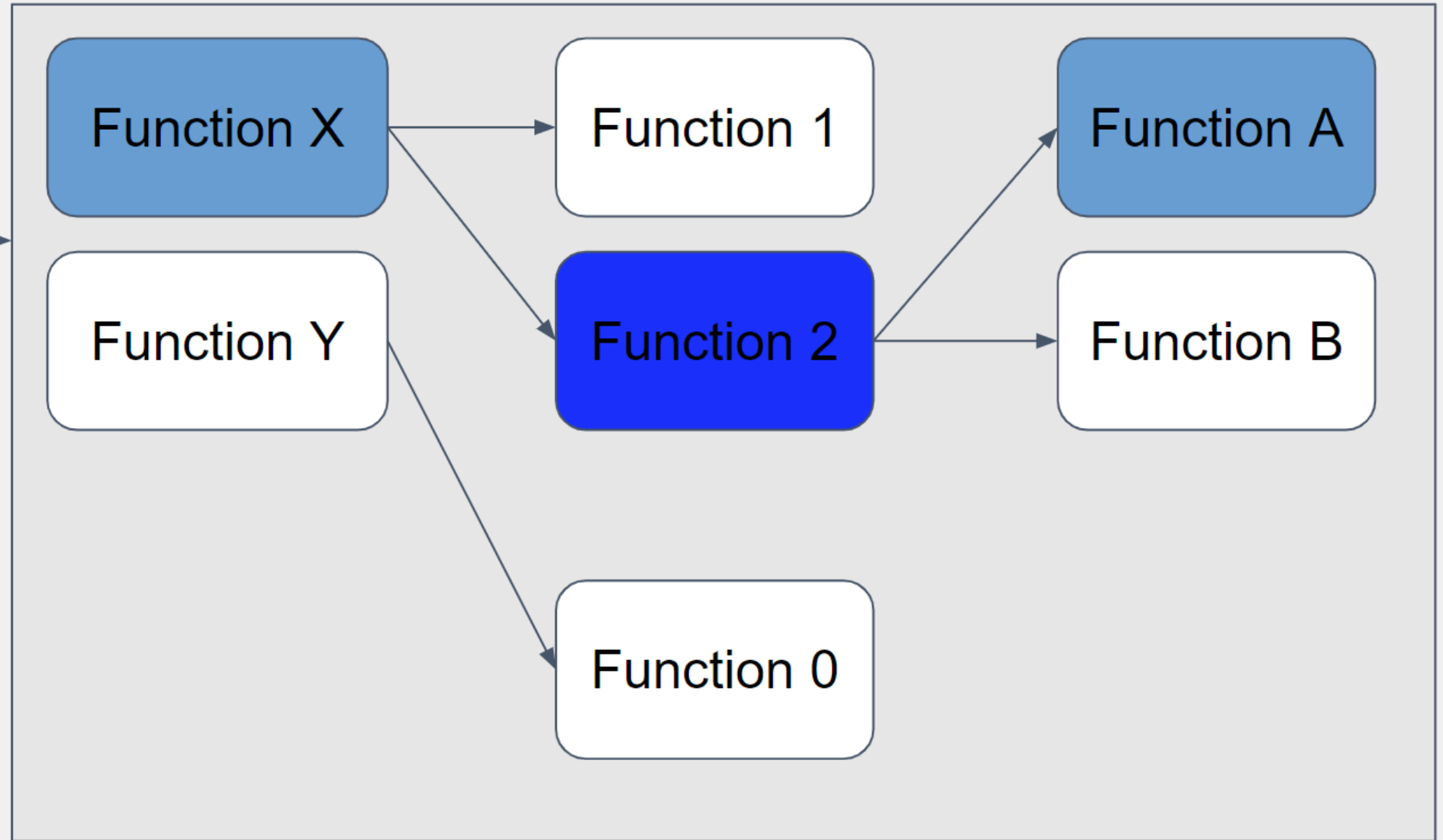
1. X
2. **X2**



# Fuzzer

Corpus:

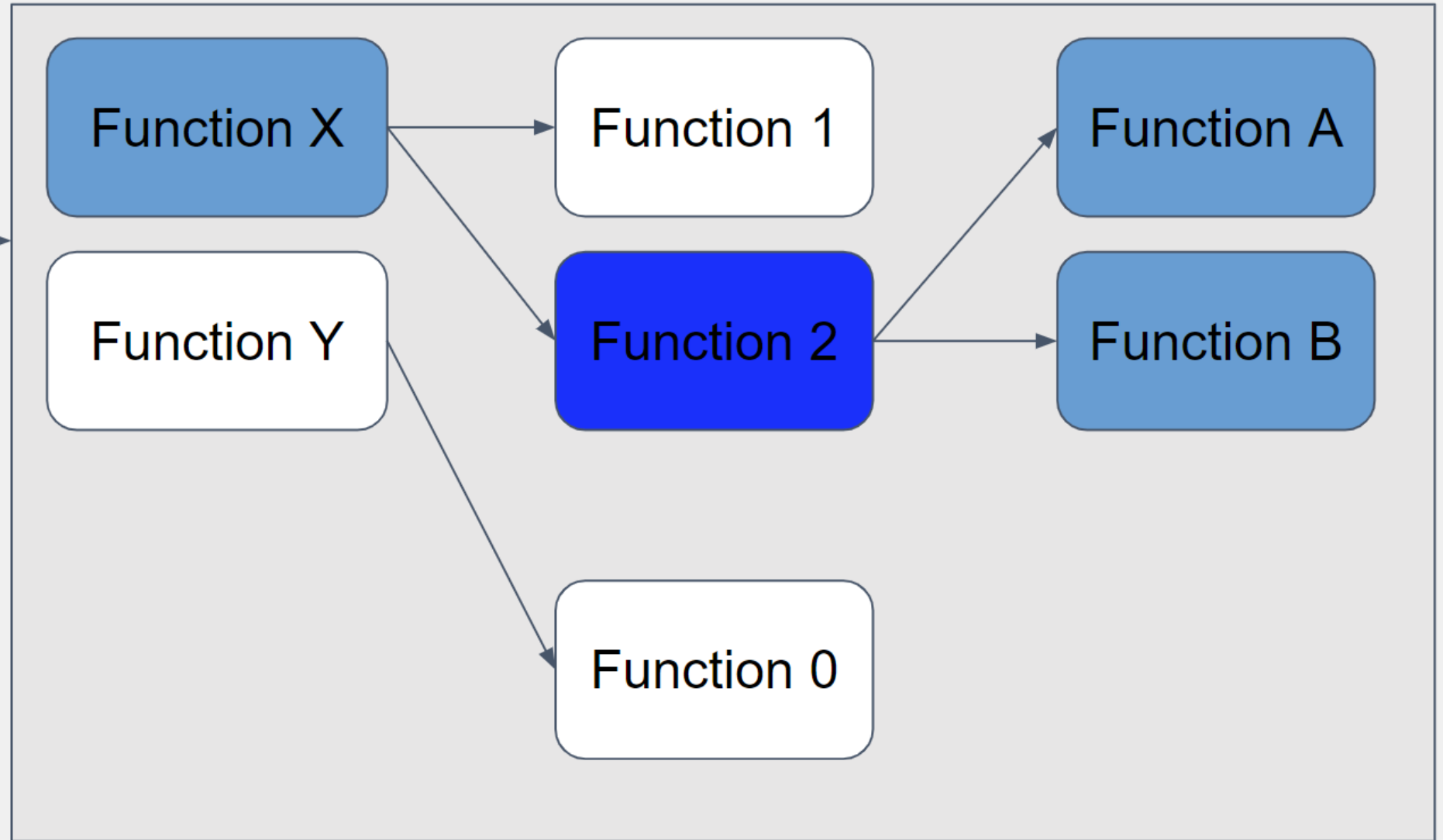
1. X
2. X2
3. X2A



# Fuzzer

Corpus:

1. X
2. X2
3. X2A
4. X2B



# More granularity than functions - basic blocks

```
w = 0;  
x = x + y;  
y = 0;  
if( x > z)  
{  
    y = x;  
    x++;  
}  
else  
{  
    y = z;  
    z++;  
}  
w = x + z;
```

Source Code

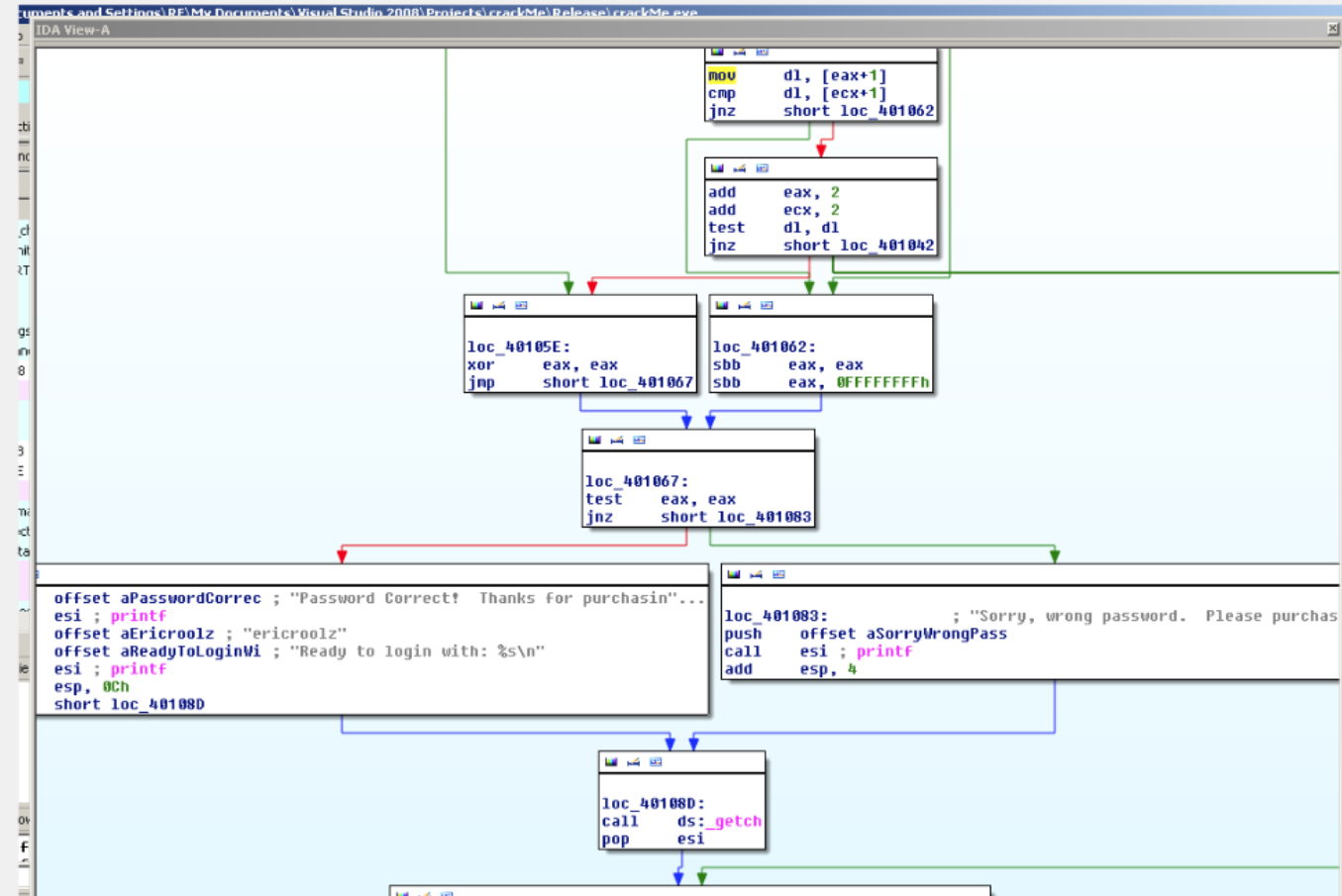
```
w = 0;  
x = x + y;  
y = 0;  
if( x > z)
```

```
y = x;  
x++;
```

```
y = z;  
z++;
```

```
w = x + z;
```

Basic Blocks



## Coverage-guiding in action

The following code wants "ABCD" input:

```
if input[0] == 'A' {  
    if input[1] == 'B' {  
        if input[2] == 'C' {  
            if input[3] == 'D' {  
                slice[input[4]] = 1 // out-of-bounds here  
            }  
        }  
    }  
}
```

Blind generation needs  $O(2^{8^4}) = O(2^{32})$  tries.

Corpus progression:

```
0. {}  
1. {"A"}  
2. {"A", "AB"}  
3. {"A", "AB", "ABC"}  
4. {"A", "AB", "ABC", "ABCD"}
```

Coverage-guided fuzzer needs  $O(4 * 2^8) = O(2^{10})$  tries.

# Basic Block based Code Coverage in AFL

- Using a "bloom filter" (byte array)
- Compare bloom filter content after every newly generated input

At each basic block, add:

```
cur_location = (block_address >> 4) ^ (block_address << 8) ;  
  
shared_mem[cur_location ^ prev_location]++;  
  
prev_location = cur_location >> 1;
```

```
if input[0] == 0x41
```

```
if input[1] == 0x42
```

```
if input[2] == 0x43
```

```
if input[3] == 0x44
```

**CRASH**

0	0	0	0	0	0
0	0	0	<b>1</b>	0	0
0	0	0	0	<b>0</b>	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	<b>0</b>	0	0	0
0	0	0	0	0	<b>0</b>

```
if input[0] == 0x41
```

```
if input[1] == 0x42
```

```
if input[2] == 0x43
```

```
if input[3] == 0x44
```

**CRASH**

0	0	0	0	0	0
0	0	0	<b>1</b>	0	0
0	0	0	0	<b>1</b>	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	<b>0</b>	0	0	0
0	0	0	0	0	<b>0</b>



```
if input[0] == 0x41
```

```
if input[1] == 0x42
```

```
if input[2] == 0x43
```

```
if input[3] == 0x44
```

**CRASH**

0	0	0	0	0	0
0	0	0	<b>1</b>	0	0
0	0	0	0	<b>1</b>	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	<b>1</b>	0	0	0
0	0	0	0	0	<b>1</b>

```

109      24 :      } else if (a == 0xFB2Au) { /* SHIN WITH SHIN DOT */
110      0 :          *ab = 0xFB2Cu;
111      0 :          found = true;
112      24 :      } else if (a == 0xFB2Bu) { /* SHIN WITH SIN DOT */
113      0 :          *ab = 0xFB2Du;
114      0 :          found = true;
115      :      }
116      35 :      break;
117      :      case 0x05BFu: /* RAFE */
118      49 :      switch (a) {
119      :      case 0x05D1u: /* BET */
120      12 :          *ab = 0xFB4Cu;
121      12 :          found = true;
122      12 :          break;
123      :      case 0x05DBu: /* KAF */
124      11 :          *ab = 0xFB4Du;
125      11 :          found = true;
126      11 :          break;
127      :      case 0x05E4u: /* PE */
128      14 :          *ab = 0xFB4Eu;
129      14 :          found = true;
130      14 :          break;
131      :      }
132      49 :      break;
133      :      case 0x05C1u: /* SHIN DOT */
134      22 :          if (a == 0x05E9u) { /* SHIN */
135      12 :              *ab = 0xFB2Au;
136      12 :              found = true;
137      10 :          } else if (a == 0xFB49u) { /* SHIN WITH DAGESH */
138      0 :              *ab = 0xFB2Cu;
139      0 :              found = true;
140      :          }
141      22 :      break;
142      :      case 0x05C2u: /* SIN DOT */
143      22 :          if (a == 0x05E9u) { /* SHIN */
144      10 :              *ab = 0xFB2Bu;
145      10 :              found = true;
146      12 :          } else if (a == 0xFB49u) { /* SHIN WITH DAGESH */
147      0 :              *ab = 0xFB2Du;
148      0 :              found = true;
149      :          }

```

Picture Source:

"Circumventing Fuzzing Roadblocks with Compiler Transformations", lafintel

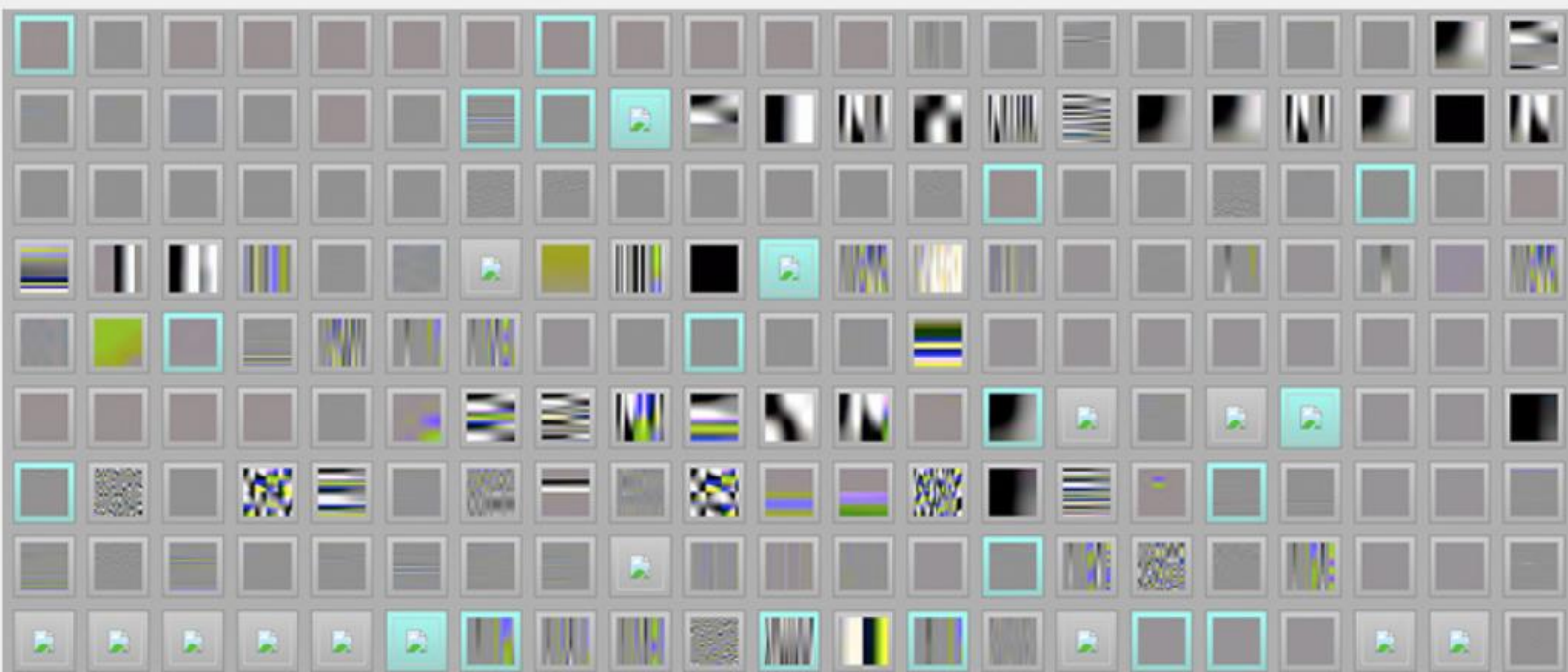


November 07, 2014

## Pulling JPEGs out of thin air

This is an interesting demonstration of the capabilities of [afl](#); I was actually pretty surprised that it worked!

```
$ mkdir in_dir  
$ echo 'hello' >in_dir/hello  
$ ./afl-fuzz -i in_dir -o out_dir ./jpeg-9a/djpeg
```



# Fuzzing Challenges

# Fuzzing Challenges

- "Bit flips" only get you this far
- afl:
  - Sequential bit flips with varying lengths and stepovers,
  - Sequential addition and subtraction of small integers,
  - Sequential insertion of known interesting integers (0, 1, INT\_MAX, UINT\_MAX, 127, 129, etc),
  - With deterministic fuzzing out of the way, the non-deterministic steps include:
    - stacked bit flips, insertions, deletions, arithmetics, and splicing of different test cases.
- Good to identify basic blocks or bugs like:

```
malloc(user_data_size) ...  
if a > 100 ...  
switch(a) ...
```

# Fuzzing Challenges

low probability of catching:

```
if a == 0x31337
```

```
if a == "CONNECT"
```

# Fuzzing Challenges

low probability of catching:

```
if a == 0x31337
```

```
if a == "CONNECT"
```

Solutions:

- wordlists
- translate into bitwise compare
- symbolic execution <3

# Fuzzing Challenges

low probability of catching:

```
if (int32) a == 0x31337
```

```
if (string) a == "CONNECT"
```

Solutions:

- wordlists
  - "CONNECT", "SEND", "RECEIVE", "OPTIONS"
  - use *strings* commands on the binary
- translate into bitwise compare
  - transform string comparison to per-byte (LD\_PRELOAD, code transformation via compiler plugin, ...)
  - if (a[0] == 0x37) { if (a[1] == 0x13) { if (a[2] == 0x03) { ...
- symbolic execution <3
  - constraint solving in code via **symbolic execution** (angr, KLEE)



# Symbolic Execution

- Translate compiled commands (assembly) into a higher-level language (e.g. VEX)
- Perform reasoning on it
- Use constraint solver to reach certain code paths
- Problems:
  - state explosion (computational power increases exponentially with code size)
  - uncertain time constraints (try to solve "if md5(input) == 0x534534534534")

**DARPA CGC**

# CGC

## DARPA Cyber Grand Challenge 2016

- Like the autonomous car challenge
- Teams create an autonomous system to attack and defend programs
  - No human interaction. Air-gapped for 2 days.
- Programs are not real x86, but a more simplistic version (only 4 system calls; read/write/exit/)
- Find bugs
  - Patch bugs in your teams computers
  - Exploit bugs in the other team computers
- Some serious HW (one rack per team, ~1000 cores, 16TB RAM)
- Finals @ Defcon Las Vegas 2016 (I was there!)

# CDC





# DARPA Grand Challenge 2004 - Self driving cars

Less well-known is the 2004 DARPA Grand Challenge, the year prior, in which no vehicle finished. In fact, no vehicle made it further than 7 miles. Most vehicles just died altogether.

Wired has a pretty neat [oral history of the 2004 DARPA Grand Challenge](#). It's short and worth a quick read.

The most impressive aspect of the 2004 race, really, is that there even was a 2005 race. After watching every vehicle fail in 2004, DARPA threw down the gauntlet again in 2005, and the rest is history.

A reporter asked, "Well, what are you gonna do?" I said, "We're gonna do it again, and this time it's going to be a \$2 million prize." It was so successful and yet so not successful, I had to do it again.

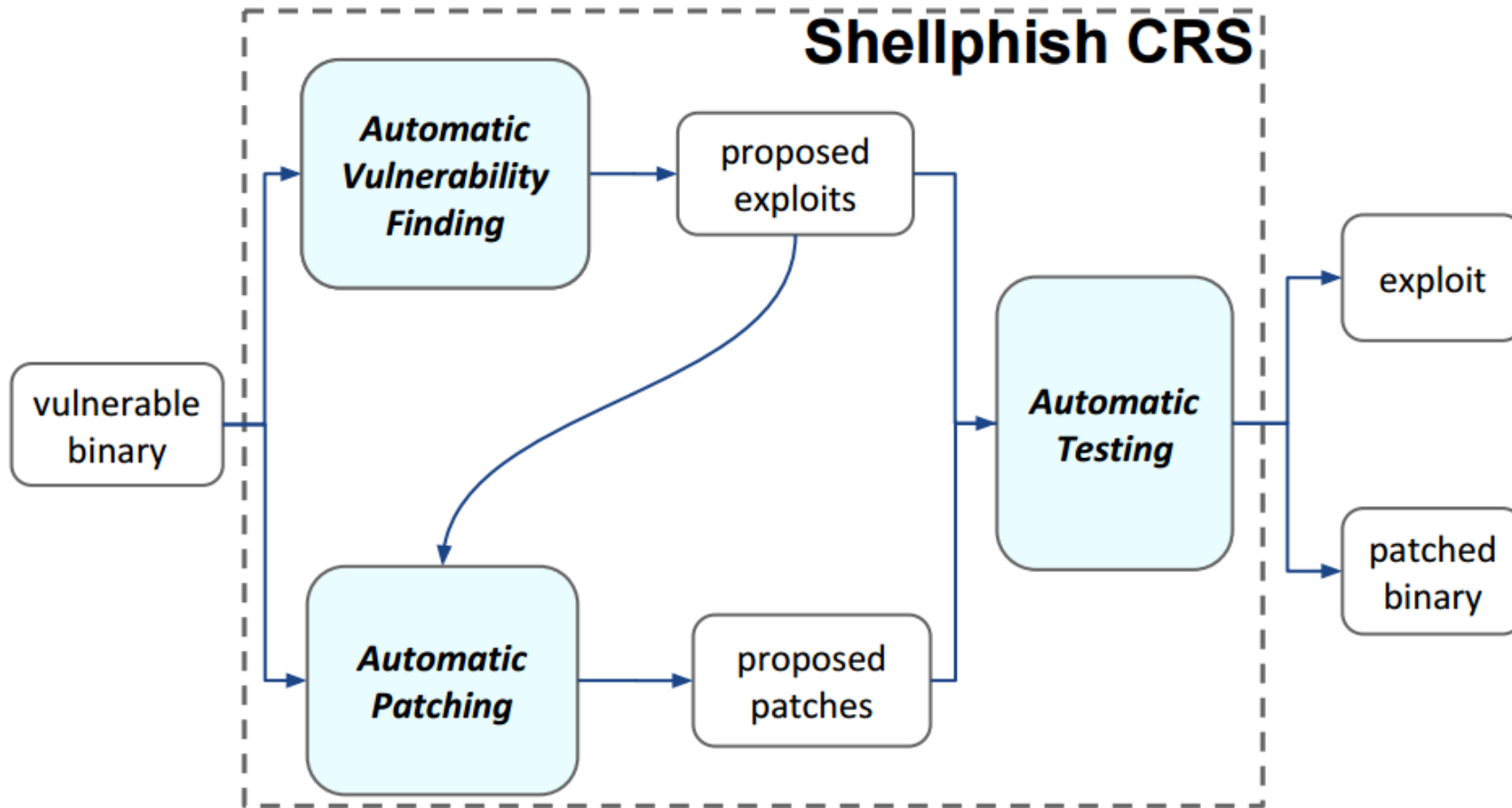
*The second competition of the DARPA Grand Challenge began at 6:40am on October 8, 2005. **All but one of the 23 finalists** in the 2005 race surpassed the 11.78 km (7.32 mi) distance **completed by the best vehicle in the 2004 race**. **Five vehicles successfully completed the 212 km (132 mi) course***



Clockwise from top left: Axion Racing's Jeep Grand Cherokee; "Sandstorm," a stripped-down self-driving Humvee from Carnegie Mellon's Red Team; Team TerraMax's Oshkosh military truck; and Team Palos Verdes High School's Acura SUV, "Doom Buggy." ALAMY

# CGC Shellphish

## Shellphish CRS



# CGC Shellphish

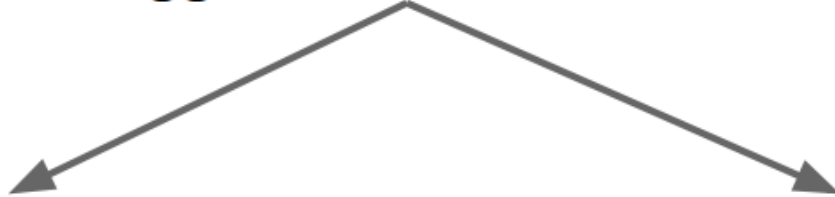
## Automatic Vulnerability Discovery



**“How do I crash a binary?”**



**“How do I trigger a condition X in a binary?”**



***Dynamic Analysis/Fuzzing***

***Symbolic Execution***

# CGC Shellphish

## Dynamic Analysis/Fuzzing



- How do I trigger the condition: "You win!" is printed?

```
x = int(input())
if x >= 10:
    if x < 100:
        → print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

- Try "1" → "You lose!"
- Try "2" → "You lose!"
- ...
- Try "10" → "You win!"



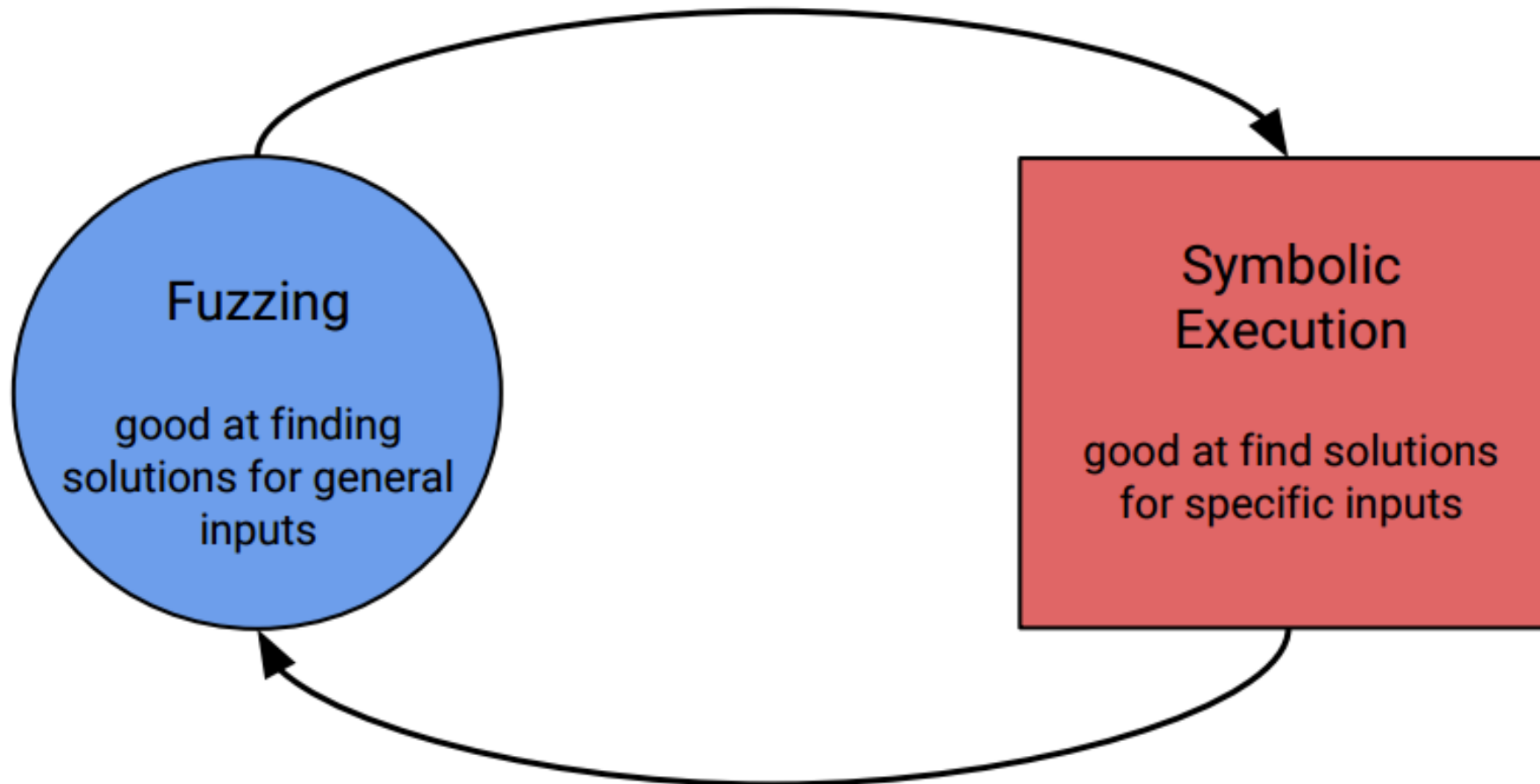


- How do I trigger the condition: "You win!" is printed?

```
x = int(input())
if x >= 10:
    if x == 123456789012:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

# CGC Shellphish

Driller = AFL + angr



# Compiler Flags

# Compiler Flags

Compiler options to enable advanced error detection routines

- GCC
- Clang

Will slow down the program massively

Will find bugs which do not directly lead to crash

Use together with fuzzing

# Compiler Flags

## AddressSanitizer (ASAN)

`-fsanitize=address`

- Fast memory error detector
- Out-of-bounds access to heap, stack, globals
- Use-after-free
- Use-after-return
- Use-after-scope
- Double free, invalid free
- For testing only (do not compile public releases with it!)

## UndefinedBehaviourSanitizer (Bsan)

`-fsanitize=undefined`

- Finds various kinds of undefined behaviour
- Null ptr, signed integer overflow, ...
- For testing only

**Other fuzzing related things...**

# Intentionally break protocols

The future:

<https://cayan.com/developers/blog-articles/how-to-protect-your-api-clients-against-breaking-c>

Roughtime is like a small “chaos monkey” for protocols, where the Roughtime server intentionally sends out a small subset of responses with various forms of protocol error

# Fuzzing closed source software

Use software instrumentation framework to inject code

- Code coverage on every basic block
- Heap overflow detector
- Etc..

Libraries:

- Intel PIN
- DynamoRIO

Fuzz parts of software via DLL's

- Korean Office Malware Identification Library...



# Fuzzing: Recap

# Fuzzing Recap

Fuzzing is:

- Finding bugs in programs
  - Especially exploitable bugs
- By bombard a program with:
  - Mutated/modified valid data
  - Generated semi-valid data

Advancements in the last few years:

- High performance code coverage measurements (identify if input triggered new code)
- Symbolic execution (use constraint solver on “if’s” to create data which lead inside the “if’s”, therefore triggering new code)

# References

<http://slides.com/revskills/fzbrowsers>

- Browser Bug Hunting and Mobile (Syscan 360)

## Shellphish:

- [http://cs.ucsb.edu/~antoniob/files/hitcon\\_2015\\_public.pdf](http://cs.ucsb.edu/~antoniob/files/hitcon_2015_public.pdf)
- <https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Shellphish-Cyber%20Grand%20Shellphish-UPDATED.pdf>

## Example:

- <https://www.evilssocket.net/2015/04/30/fuzzing-with-afl-fuzz-a-practical-example-afl-vs-binutils/>