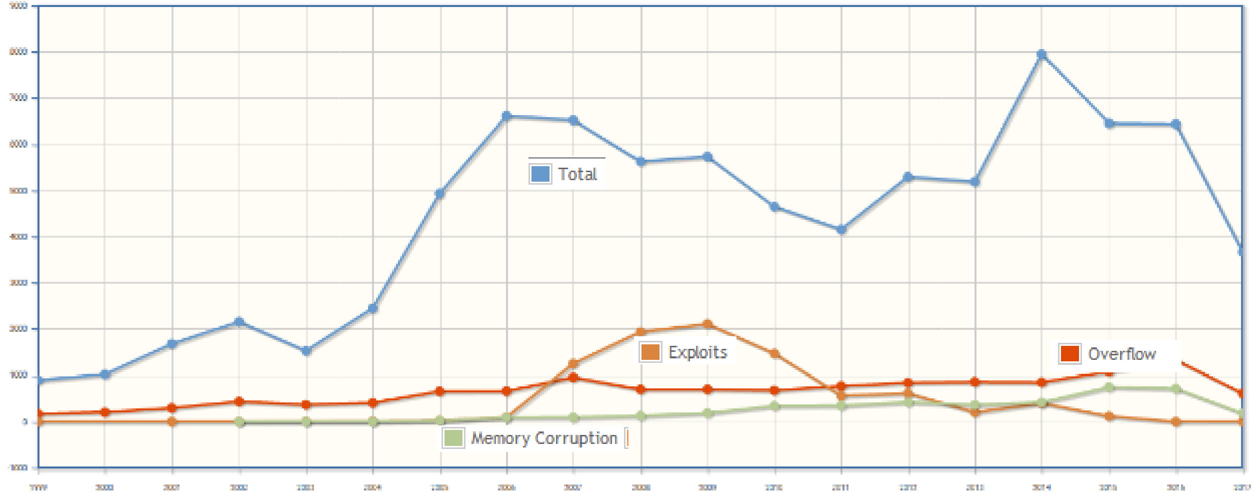
Introduction to Fuzzing

Sanjay Rawat

sanjay.rawat@Bristol.ac.uk

Why do we care?



Vulnerabilities by type & year (http://www.cvedetails.com)

Organization

- Memory corruption vulnerabilities
- Exploitability- attack model
- Fuzzing- finding vulnerabilities
- Types of Fuzzing
- Limitations/challenges
- Some existing solutions

Memory Corruption Vulnerabilities

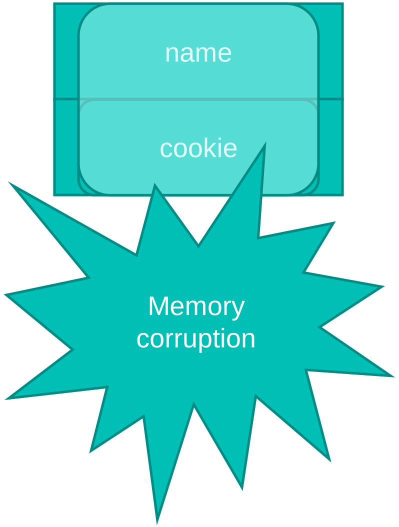
- WYSINWYX: What You See Is Not What You eXecute by G. Balakrishnan et. al.
  - Higher level code -> low-level representation
  - Seemingly separate variables -> contiguous memory addresses

- Contiguous memory locations allow for boundary violations!

example

```
#include <stdio.h>
int get_cookie(){
  return rand();}
int main(){
    int cookie;
    char name[40];
    cookie = get_cookie();
    gets(name);
    if (cookie == 0x41424344)
        printf("You win %s\n!", name);
    else printf("better luck next time :(");
    return 0;

}
```



Side effects

- Over/underflow
- Sensitive data corruption
- Control data corruption (control hijacking)

If done properly-exploit

Otherwise crash!

Fuzzing

- Run program on many **abnormal/malformed** inputs, look for **unintended** behavior, e.g. crash.

- Underlying assumption: *if the unintended behavior is dependent on input, an attacker can craft such an input to exploit the bug.*

Types of Fuzzing

- Input based: mutational and Generative
- Application based: black-box and white-box
- Strategy: memory-less and evolutionary

Input Generation

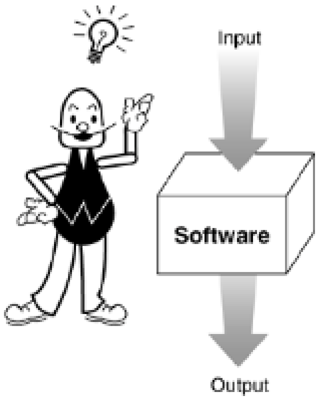
- Mutation Based: mutate seed inputs to create new test inputs

- Generation Based: Learn/create the format/model on the input and based on the learned model, generate new inputs

Application Monitoring

- Blackbox: Only interface is known.

- Whitebox: Application internals are known.



Problem with Traditional Fuzzing

Blackbox fuzzing+mutation: Aiming with luck!

```
... //JPEG parsing

read(fd, buf, size);

if (buf[1] == 0xD8 && buf[0] == 0xFF)

  // interesting code here

else
  pr_exit(“Invalid file”);
```

Problem with Traditional Fuzzing

- Apply more heuristics to:
  - Mutate better
  - Learn good inputs

- Apply more analysis (static/dynamic) to understand the application behavior

Problem with ~~Traditional~~ Smart Fuzzing

smart fuzzing: Aiming with educated guess!

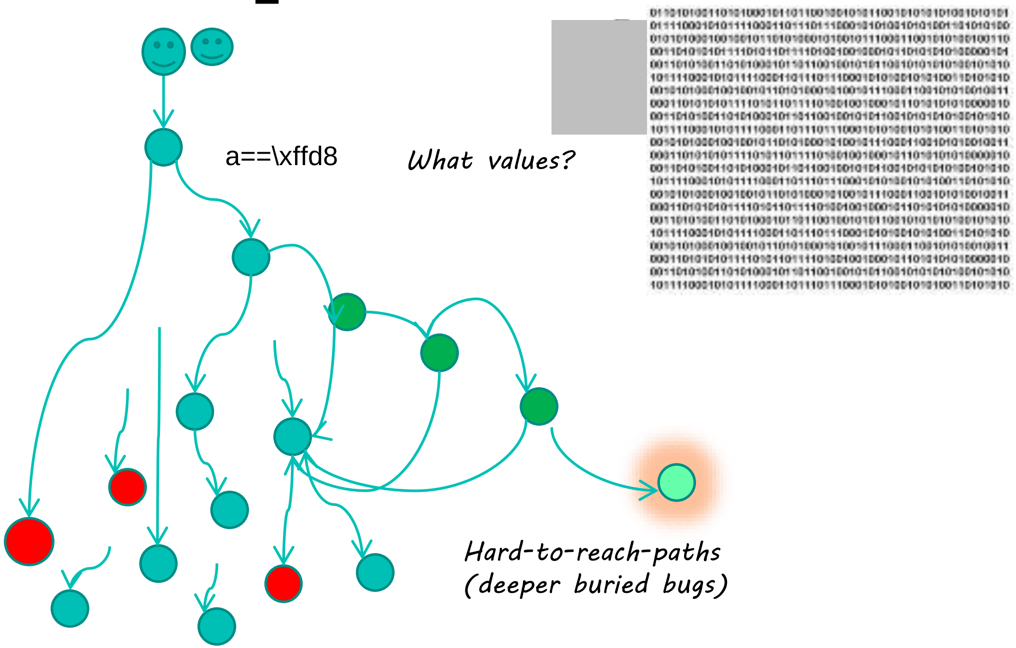
Evolutionary Fuzzing

- Recall: memory-less and Evolutionary fuzzing
- Rather than throwing inputs, *evolve them.*
- *Underlying assumption:*
  - *Inputs are parsed enough before going further deeper in execution*

But, there are issues

Problem Exemplified….

Where is ‘a’?



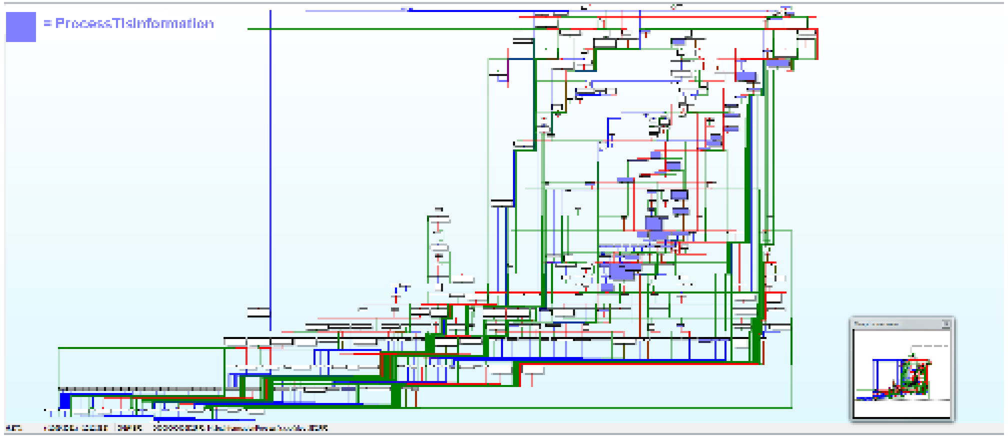
Easy paths (superficial paths), error code

Issues identified…

- For smart code-coverage based fuzzer, it is important to have some knowledge about:
  - Where (which offsets in input) to apply mutation
  - What values to replace with.
  - How to avoid traps (paths leading to error handling code)

Fuzzing+Symbex

- Symbolic/concolic execution can answer such questions.
  - *Driller*: Augmenting *Fuzzing* Through Selective Symbolic Execution, NDSS'16
- But... Scalability?



Recent Observations on Fuzzing

- Lava: Large-scale automated vulnerability addition,” IEEE S&P ’16.
  - large numbers of realistic bugs into program source code.
  - Results are not very encouraging for fuzzing!

Recent Observations on fuzzing+Symbex

- Recall LAVA results on symbex
- Experience Report: How is Dynamic Symbolic Execution Different from Manual Testing? – A Study on KLEE, In: ISSTA'15.
  - Manually developed test suites perform better than KLEE-based test suites on covering hard-to-cover code and killing hard-to-kill mutants.
  - KLEE-based test suites are less effective on exploring some meaningful paths and generating valid string structural inputs to go through the input parser.

Concrete results (From LAVA paper)

Program	Total Bugs	Unique Bugs Found		
		FUZZER	SES	Combined
`uniq`	28	7	0	7
`base64`	44	7	9	14
`md5sum`	57	2	0	2
`who`	2136	0	18	18
Total	2265	16	27	41

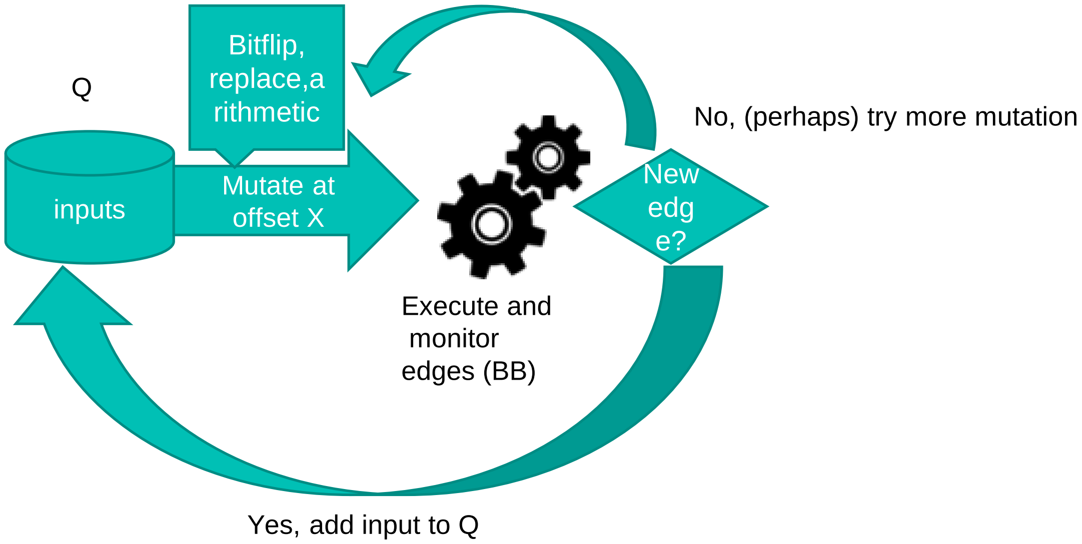
What more.. from the author of LAVA…
http://moyix.blogspot.nl/2016/07/fuzzing-with-afl-is-an-art.html

"… Because I'm lucky enough to have a 24 core server sitting around, I gave it 24 cores (one using -M and the rest using -S) and let it run for about 4 and a half days, fully expecting that it would find the input in that time.

This did not turn out so well… "

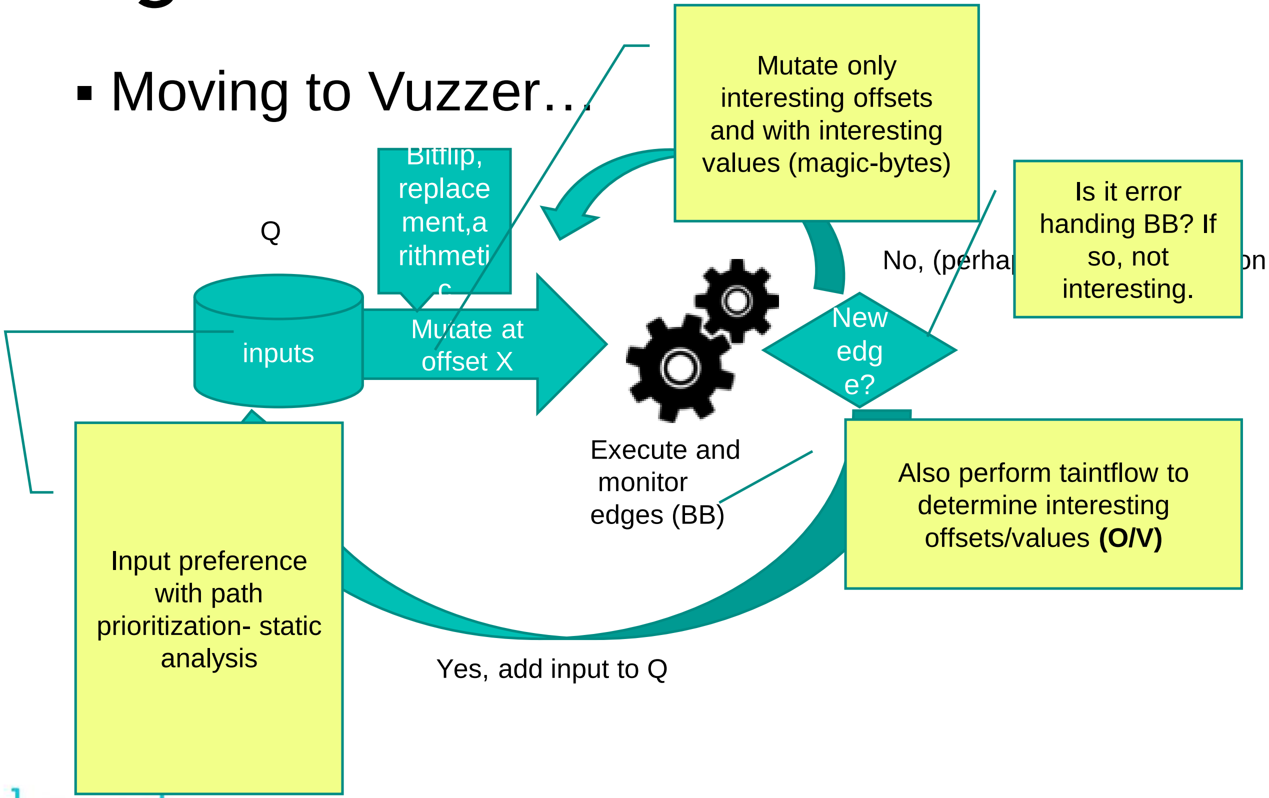
Evolving A Fuzzer

- Lets start with something we know- AFL

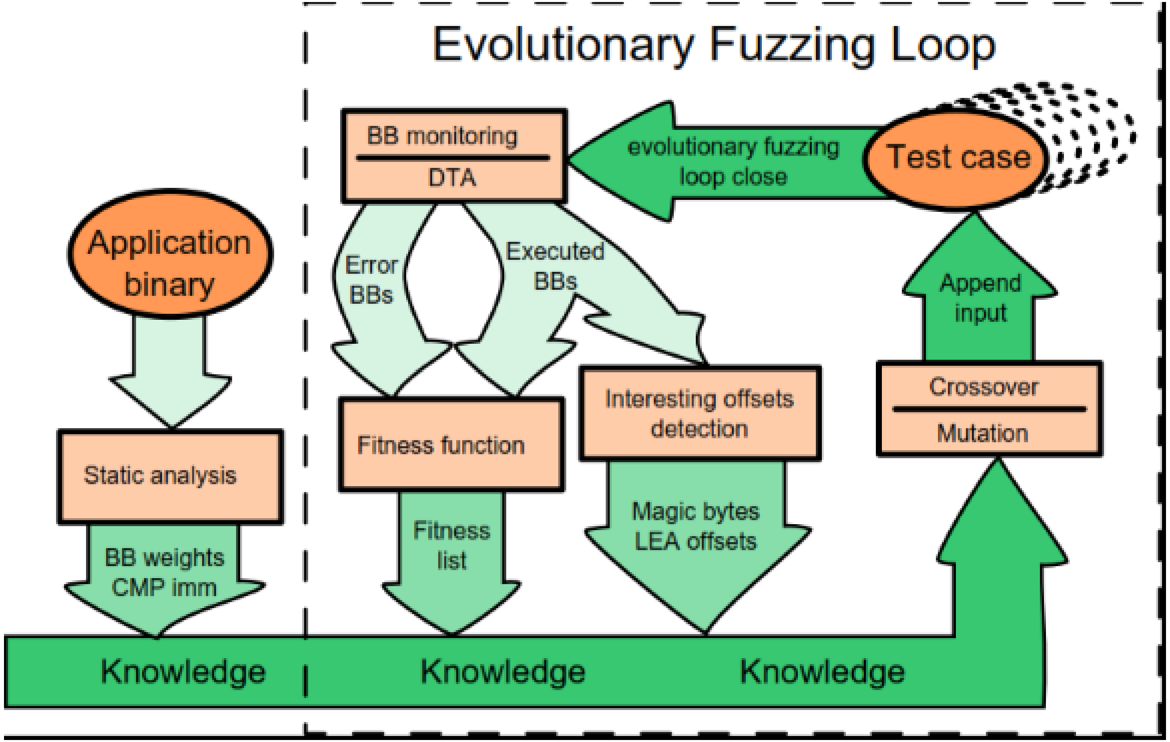


Evolving Taintflow based Solution

- Moving to Vuzzer…



Our Solution: Vuzzer (NDSS'17)



Concurrency Bugs Detection via Fuzzing

- Concurrency bugs are caused by non-deterministic interleavings between shared memory accesses.
- Their effects propagate through data and control dependences until they cause software to crash, hang, produce incorrect output, etc
- Interleavings are not only complicated to reason about, but they also dramatically increase the state space of software.

Fuzzing the scheduler

- Identify shared objects
- An input that executes instructions involving shared objects
- Thread schedular
  - Rather than letting OS decides, introducing a schedular that can control the thread scheduling
  - Schedule threads w.r.t. different ordering

Razzer: Kernel race bug detection

- In: 2019 IEEE Symposium on Security and Privacy (SP)
- Involves static and dynamic analysis
- Found 30 new race bugs in the latest kernel
- Main Idea:
  - Find shared interleaved objects
  - Find an input (by fuzzing) that hits a race (in single thread)
  - Use the input for fuzzing the interleaving of thread in kernel

1. Static analysis component

- Identifying Race candidates ($\texttt{RacePair}_{\texttt{cand}}$)
  - Instructions that access (points) to the same memory location
  - Point-to analysis
  - Difficult to get it right!
    - Interprocedural analysis
    - Conservative
    - Partition based analysis (scalability)
    - Rather than analysing the entire kernel code, it partition the space w.r.t. directory structure, e.g. `Kernel, mm, fs, drivers`

2. Scheduler in Hypervisor

- Running the Razzer on a tailored VM
  - Fuzzing mulit-threaded program in guest user-land
  - Triggering races in guest OS
- Uses Virtual Machine Control Structure to:
  - Set hardware breakpoints
  - To catch when the interrupt occurs
- Resume per-Core Execution
  - At each breakpoint, ability to decide which thread to resume.

3. Two-phase fuzzing

- Single-Thread Fuzzing
  - User program generation (Single thread)
    - Random sequences of syscalls with random values of parameter
    - It uses Syzkaller
  - User program execution (single thread)
    - Execute the above program and monitors (kcov)
      - Checks if two syscalls execute addresses related to a single $\texttt{RacePair}_{\texttt{cand}}$
    - It annotates such syscalls with the corresponding addresses from $\texttt{RacePair}_{\texttt{cand}}$

3. Two-phase fuzzing conti…

- Multi-Tread generator
  - Creates a multi-thread version of the single-thread user program
    - If the annotated syscalls are i, j

```
# Get pinned threads, thr0 and thr1
thr0 = get_pinned_thread(vCPU0)
thr1 = get_pinned_thread(vCPU1)

# Assign syscalls to thr0 and thr1
syscalls = get_syscalls(Pst)
thr0.add_syscalls(syscalls[:i])
thr1.add_syscalls(syscalls[i+1:j])

# Determine the execution order
r = random([vCPU0, vCPU1])
thr0.add_hypercall(hcall_order(r))

# Trigger and check races
thr0.add_hypercall(hcall_set_bp(vCPU0, RP_i))
thr0.add_syscalls(syscalls[i])
thr0.add_hypercall(hcall_check-race())

thr1.add_hypercall(hcall_set_bp(vCPU1, RP_j))
thr1.add_syscalls(syscalls[j])
thr1.add_hypercall(hcall_check_race())
```

3. Two-phase fuzzing conti…

- Multi-Thread Executor
  - Sets breakpoints at addresses in $\texttt{RacePair}_{\texttt{cand}}$
  - Checks if the breakpoints are hit
  - Concrete addresses pointed to by the respective instructions
- Several address sanitizers are enabled during the kernel compilation.