CSC 591
Systems Attacks and Defences

Symbolic Execution

Alexandros Kapravelos
akaprav@ncsu.edu

Let's find some bugs

- We have a potentially vulnerable program
- The program has some inputs which can be controlled by the attacker
- What should we do as developers?
  - Add checks (assertions)
  - Write tests
  - Make sure the checks do not fail

- Is this enough?

Concrete Execution

```
void foo(int x, int y) {
    int z = 0;
    if (x > y) {
        z = x;
    }
    else {
        z = y;
    }
    if (z < x) {
        assert false;
    }
}
```

x=0, y=0

False

z = 0

False

// not reached

Concrete Execution

```
void foo(int x, int y) {
    int z = 0;
    if (x > y) {
        z = x;
    }
    else {
        z = y;
    }
    if (z < x) {
        assert false;
    }
}
```

x=1, y=0

True

z = 1

False

// not reached

Pros/Cons

- Testing intended functionality
- Testing for known bugs
- Unintended functionality
- Unknown bugs
- Complete coverage

Can we automate this part?

Symbolic Execution

```
void foo(int x, int y) {
    int z = 0;
    if (x > y) {
        z = x;
    }
    else {
        z = y;
    }
    if (z < x) {
        assert false;
    }
}
```

x=α, y=β

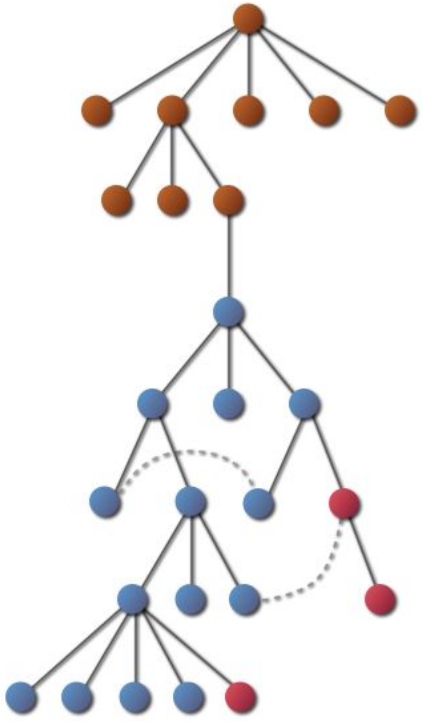
z = α, α > β

z = β, α <= β

1. α < α -> False
2. β < α, α <= β, False

Feasible and Infeasible Paths

- A path is a particular route in the control-flow graph of the program
- A **feasible** path is the path covered for a particular input
- An **infeasible** path is the path that no input can cover



Infeasible Paths

- Dead code => infeasible path
- Infeasible path !=> dead code

- It is normal in a large program to have a large number of infeasible paths

- This makes automatic testing based on the input to the program incredibly hard

Constrains

- $\alpha > \beta \land \alpha + \beta <= 10$

- α, β are called **free variables**
- Solution: a set of variable assignments that makes the constraint satisfiable
- $\{\alpha = 3, \beta = 2\}$ is a solution
- $\{\alpha = 6, \beta = 5\}$ is not a solution

- Decision procedure: is the constraint satisfiable?
- Constraint solver: if is satisfiable, find assignments
- Undecidable problem

Symbolic Execution

- Execute the program differently, “symbols” as input
- Take all feasible paths
- Program state is different:
  - No stack/heap
  - Symbolic values for memory locations
  - Path condition
- Path condition: input constraints so that a certain path is feasible
- A solution to a path condition is a test input that covers the desired path

History of Symbolic Execution

James C. King

Symbolic execution and program testing

Communications of the ACM

(July 1976)

Why are we talking about it now?

- Computation intensive
  - Too many paths
  - Program state grows a lot
  - Constraint solver is computationally expensive, but we need to identify the feasible paths

- Powerful computers
- Better constraint solvers

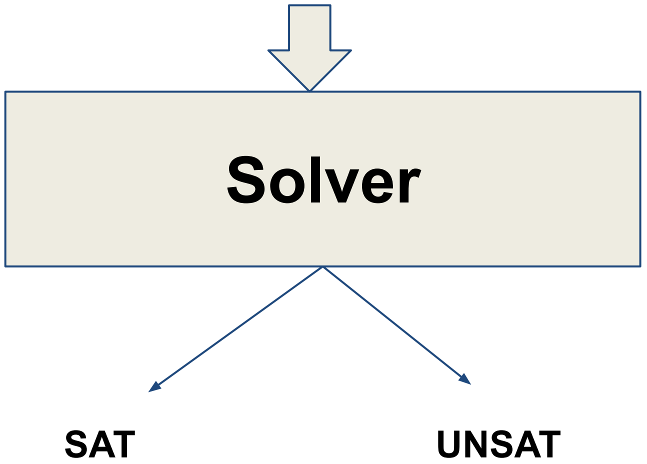
Symbolic Execution Tools

- KLEE
  - Open source symbolic executor
  - Runs on top of LLVM
  - Has found lots of problems in open-source software
- SAGE
  - Microsoft internal tool
  - Symbolic execution to find bugs in file parsers - E.g., JPEG, DOCX, PPT, etc
  - Cluster of n machines continuously running SAGE

Constraint Solver

- Boolean **SAT**isfiability Problem
- Find values that satisfy a boolean formula
- NP-Complete

$$(l1 \lor l2 \lor x2) \land (\neg x2 \lor l3 \lor x3)$$



SMT Solvers

- Satisfiability modulo theories
- SAT, but with binary variables replaced by predicates over a suitable set of non-binary variables

3x + 2y - z >= 4

$$(\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geq 2.3y) \wedge \left(\neg b \vee y < -34.4 \vee \exp(x) > \frac{y}{x}\right)$$

Popular SMT solvers

- Z3 - developed at Microsoft Research
  - https://github.com/Z3Prover/z3
- Yices - developed at SRI
  - http://yices.csl.sri.com/
- STP - developed by Vijay Ganesh, now @ Waterloo
  - http://stp.github.io/
- CVC3 - developed primarily at NYU
  - http://www.cs.nyu.edu/acsys/cvc3/

Forking Execution

- What to do when we reach a branching point?
  - Follow both paths (condition satisfied and negation)
- State explosion *really* fast (exponential)
  - Loops on symbolic variables are problematic
- How can we do this more efficiently?
  - Prune paths by following only feasible ones
  - Concolic execution: run the program concretely and assist the execution with symbolic execution by changing the path conditions

Static analysis

- It will terminate, even if the whole program is taken into account
- Approximation is the key
  - Let's assume every path is feasible
- False alarms
- Less accurate

Symbolic search

- We have to decide on a strategy
  - Depth-first search (DFS)
  - Breadth-first search (BFS)

- Potential drawbacks
  - No smart choices
  - DFS can get easily stuck in one part of the program
    - Literally on a loop
  - BFS is a better choice
    - Harder to implement (think about concolic execution)

Search strategies

- Focus on the paths that matter
  - Assertion failures
  - Time bound
  - Vulnerable functions (like strcmp)

- Improve coverage
  - Program execution as a DAG
    - Nodes = program states
    - Edge(n1, n2) = can transition from n1 to state n2
  - Graph exploration algorithm

Randomness

- In the beginning we know nothing, how do we start?
- Ideas
  - Pick next path at random
  - Randomly restart search
  - Choose randomly among equal priority paths

- But then how do we reproduce our analysis?
  - Pseudo-randomness
  - Record the seed
  - Otherwise bugs can disappear on reruns

Coverage-guided heuristics

- Let's visit statements that we haven't seen before
- Approach
  - Score of statement = # visits
  - Pick the next statement with the lowest score
- Pros
  - Errors are often in hard-to-reach parts of the program
  - This strategy tries to reach everywhere.
- Cons
  - Maybe never be able to get to a statement if proper precondition not set up

Generational search

- Hybrid of BFS and coverage-guided
  - Generation 0: pick one program at random, run to completion
  - Generation 1: take paths from gen 0; negate one branch condition on a path to yield a new path prefix; find a solution for that prefix; then take the resulting path
  - Generation n: similar, but branching off gen n-1
- Also uses a coverage heuristic to pick priority

Path-based search limited

```
int counter = 0, values = 0;
for (i = 0; i<100; i++) {
  if (input[i] == 'B') {
    counter++;
    values += 2;
  }
}
assert(counter != 75);
```

- 2^{100} possible execution paths
- Hard to find the bug
  - $\binom{100}{75} \approx 2^{78}$ paths reach buggy line of code
  - $Pr(\text{finding bug}) = 2^{78} / 2^{100} = 2^{-22}$

Libraries and native code

- Execution of a program is not solely contained on the program's code
  - Libraries, system calls, assembly code
- We could extend the symbolic execution to those parts
  - Pull in the library and symbolically execute it
  - If library is complicated, then our program state will grow too large
  - Replace the library with a simpler version (libc -> newlib)
- Model the code of the external dependencies

Concolic Execution

- Dynamic symbolic execution
- Concrete execution of the program with assistance by symbolic execution

- Instrument the program
  - Keep a shadow state with symbolic variables
  - Start with a concrete execution that sets an initial path

- Follow one path and use symbolic execution to determine the next one
  - Negate a condition
  - Inputs are concrete values

Concretization

- Use symbolic execution as guidance
  - But replace symbolic variables with concrete values that satisfy the path condition
- This way the program is actually executed
  - Abstract parts that are not in the code (system calls)
  - No symbolic-ness at such calls (we lose information)
- Very useful when conditions get too complex for SMT solver

Conclusion

- Symbolic execution is very powerful and productive
- Not very practical as programs grow large
  - Limited by the power of the constraint solver
  - Bound by the infeasible paths number
- Promising research area!

CSC 591
Systems Attacks and Defenses

Fuzzing

Alexandros Kapravelos
akaprav@ncsu.edu

Let’s find some bugs (again)

- We have a potentially vulnerable program
- The program has some inputs which can be controlled by the attacker

Can we generate automatic tests?

Fuzzing

- A form of vulnerability analysis
- Steps
  - Generate random inputs and feed them to the program
  - Monitor the application for any kinds of errors

- Simple technique
- Inefficient
  - Input usually has a specific format, randomly generated inputs will be rejected
  - Probability of causing a crash is very low

Example

Standard HTML document

- <html></html>

Randomized HTML

- <html>AAAAAAA</html>
- <html><></html>
- <html></html></html>
- <html>html</html>
- <html>/</<>></html>

Types of Fuzzers

- **Mutation Based**
  - mutate existing data samples to create test data
- **Generation Based**
  - define new tests based on models of the input
- **Evolutionary**
  - Generate inputs based on response from program

Mutation Based Fuzzing

- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
- Anomalies may be completely random or follow some heuristics
- Requires little to no setup time
- Dependent on the inputs being modified
- May fail for protocols with checksums, those which depend on challenge response, etc.

- Example Tools:
  - Taof, GPF, ProxyFuzz,
  - Peach Fuzzer, etc.

Fuzzing a pdf viewer

- Google for .pdf files (about 1,640,000,000 results)
- Crawl pages and build a pdf dataset
- Create a fuzzing tool that:
  - Picks a PDF file
  - Mutates the file
  - Renders the PDF in the viewer
  - Check if it crashes

Mutation Based Fuzzing

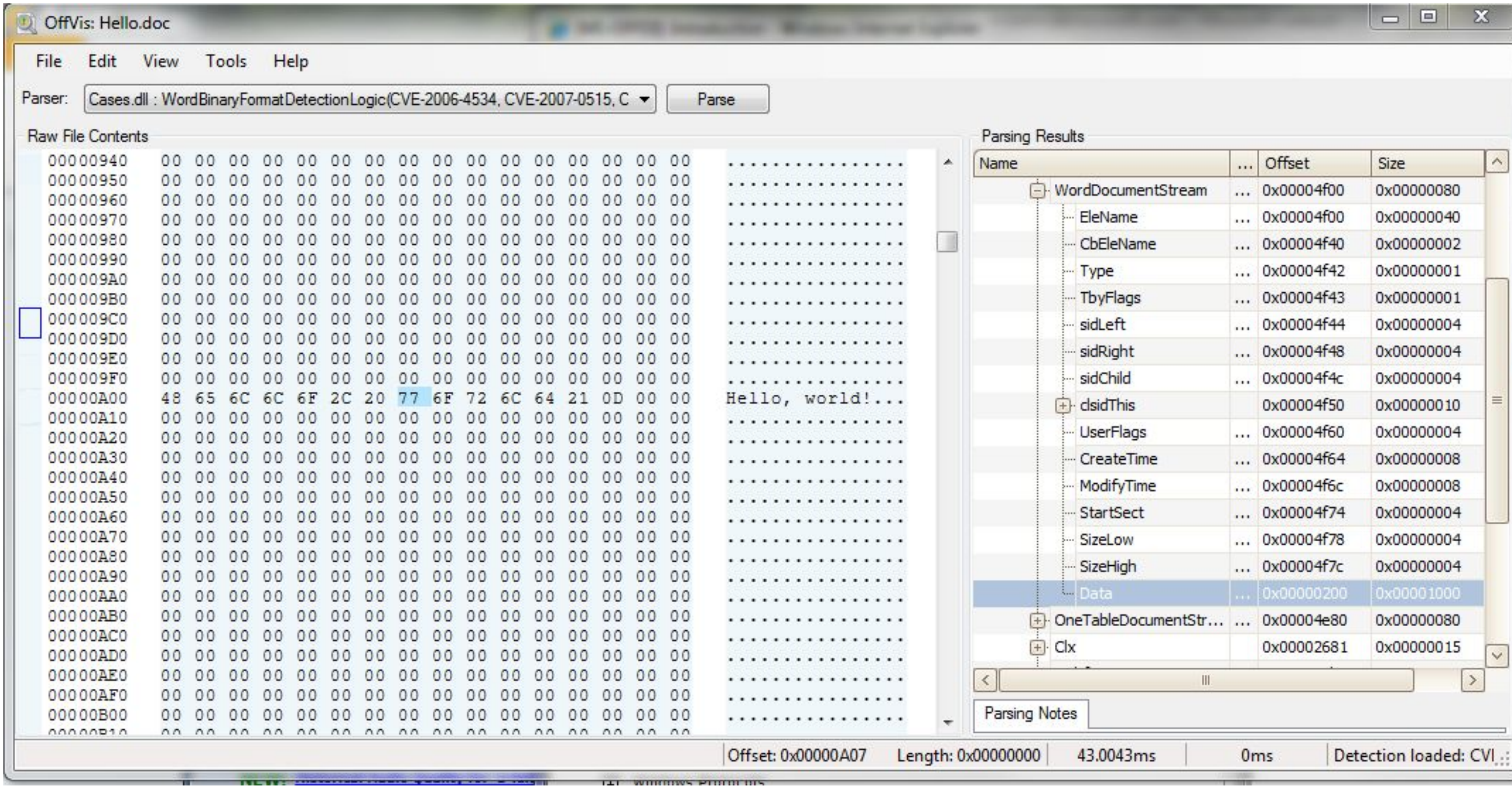
- East to setup and automate
- Little to no protocol knowledge required

- Limited to the initial dataset
- May fail on protocols with checksums, or other challenges

Generation-Based Fuzzing

- Generate random inputs with the input specification in mind (RFC, documentation, etc.)
- Add anomalies to each possible spot
- Knowledge of the protocol prunes inputs that would have been rejected by the application

Word (.doc) Binary File Format



https://msdn.microsoft.com/en-us/library/office/cc313105(v=office.14).aspx (576 pages)

Generation-Based Fuzzing

- Completeness
- Can deal with complex input, like checksums

- Input generator is labor intensive for complex protocols
- There has to be a specification

Evolutionary Fuzzing

- Attempts to generate inputs based on the response of the program

- Autodafe
  - Fuzzing by weighting attacks with markers
  - Open source

- Evolutionary Fuzzing System (EFS)
  - Generates test cases based on code coverage metrics

Challenges

- Mutation based
  - Enormous amount of generated inputs
  - Can run forever
- Generation based
  - Less inputs (we have more knowledge)
  - Is it enough?

Code Coverage

- A metric of how well your code was tested
- Percent of code that was executed during analysis
- Profiling tools
  - gcov

- Code coverage types:
  - Line coverage
    - which lines of source code have been executed
  - Branch coverage
    - which branches have been taken
  - Path coverage
    - which paths were taken

Fuzzing Chrome

- AddressSanitizer
- ClusterFuzz
- SyzyASAN
- ThreadSanitizer
- libFuzzer
- more...

Chrome's fuzzing infrastructure

- Automatically grab the most current Chrome LKGR (Last Known Good Revision)
- Hammer away at it to the tune of multi-million test cases a day
- Thousands of Chrome instances
- Hundreds of virtual machines

AddressSanitizer

- Compiler which performs instrumentation
- Run-time library that replaces malloc(), free(), etc
- custom malloc() allocates more bytes than requested and “poisons” the redzones around the region returned to the caller

- Heap buffer overrun/underrun (out-of-bounds access)
- Use after free
- Stack buffer overrun/underrun

- Chromium’s “browser_tests” are about 20% slower

AddressSanitizer Results

- 10 months of testing the tool with Chromium (May 2011)
- 300 previously unknown bugs in the Chromium code and in third-party libraries
  - 210 bugs were heap-use-after-free
  - 73 were heap-buffer-overflow
  - 8 global-buffer-overflow
  - 7 stack-buffer-overflow
  - 1 memcpy parameter overlap
- 1.73x performance penalty

SyzyASAN

- AddressSanitizer works only on Linux and Mac
- Different instrumenter that injects instrumentation into binaries produced by the Microsoft Visual Studio toolchain
- Run-time library that replaces malloc, free, et al.
- ~4.7x performance penalty

ThreadSanitizer

- Runtime data race detector based on binary translation
- Supports also compile-time instrumentation
  - Greater speed and accuracy
- Data races in C++ and Go code
- Synchronization issues
  - deadlocks
  - unjoined threads
  - destroying locked mutexes
  - use of async-signal
  - unsafe code in signal handlers
  - Others…
- ~5x-15x performance penalty

libFuzzer

- Engine for in-process, coverage-guided, whitebox fuzzing
- In-process
  - don’t launch a new process for every test case
  - mutate inputs directly in memory
- Coverage-guided
  - measure code coverage for every input
  - accumulate test cases that increase overall coverage
- Whitebox
  - compile-time instrumentation of the source code
- Fuzz individual components of Chrome
  - don’t need to generate an HTML page or network payload and launch the whole browser

libFuzzer

```
==9896==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x62e000022836 at
pc 0x000000499c51 bp 0x7fffa0dc1450 sp 0x7fffa0dc0c00
WRITE of size 41994 at 0x62e000022836 thread T0
SCARINESS: 45 (multi-byte-write-heap-buffer-overflow)
   #0 0x499c50 in __asan_memcpy
   #1 0x4e6b50 in Read third_party/woff2/src/buffer.h:86:7
   #2 0x4e6b50 in ReconstructGlyf third_party/woff2/src/woff2_dec.cc:500
   #3 0x4e6b50 in ReconstructFont third_party/woff2/src/woff2_dec.cc:917
   #4 0x4e6b50 in woff2::ConvertWOFF2ToTTF(unsigned char const*, unsigned long,
woff2::WOFF2Out*) third_party/woff2/src/woff2_dec.cc:1282
    #5 0x4dbfd6 in LLVMFuzzerTestOneInput
   testing/libfuzzer/fuzzers/convert_woff2ttf_fuzzer.cc:15:3
```

Cluster Fuzzing

ClusterFuzz uses the following memory debugging tools with libFuzzer-based fuzzers:

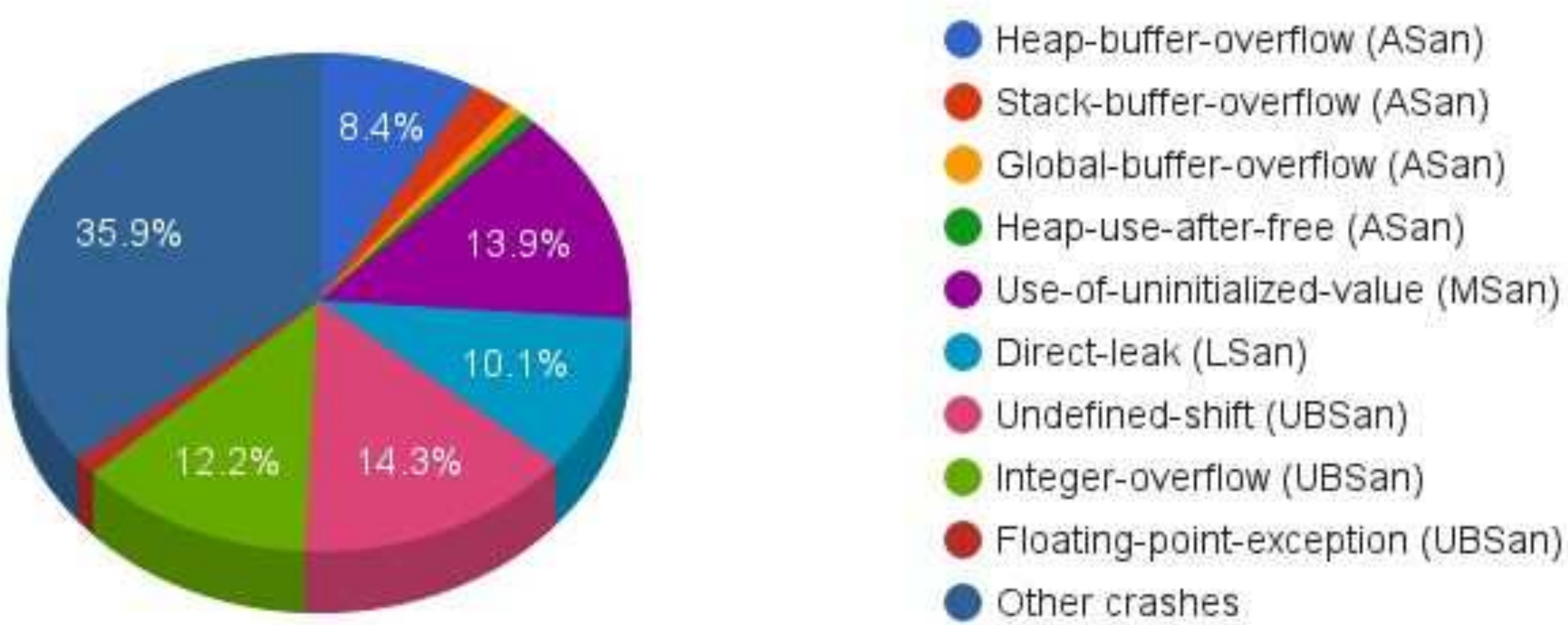
- AddressSanitizer (ASan): 500 GCE VMs
- MemorySanitizer (MSan): 100 GCE VMs
- UndefinedBehaviorSanitizer (UBSan): 100 GCE VMs

July 2016 (30 days of fuzzing)

14,366,371,459,772 unique test inputs

112 bugs filed

Analysis of the bugs found so far



Source: https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html

Chrome’s Vulnerability Reward Program

- Submit your fuzzer
- Google will run it with ClusterFuzz
- Automatically nominate bugs they find for reward payments