Fuzzing from First Principles

Alisa Esage
Zero Day Engineering LLC
Off By One Security Podcast, September 2024

Reality check

Google is continuously fuzzing its own software products using a broad array of **professionally maintained state-of-the-art fuzzers** on a distributed fuzzing cloud with **tens of thousands fuzzing instances** (more?)

Researchers **consistently** come up with new 0-days in v8 and Chrome

And they find it by… fuzzing?!

Smart Fuzzing

🚫 Not necessarily:

- Scaling
- Parallelization
- Code coverage guidance
- Augmenting with concolic execution
- Using the latest cool tool
- Assisting fuzzing with specialized hardware features
- …

✅ Ultimately, it comes down to:

1. Knowledge from First Principles

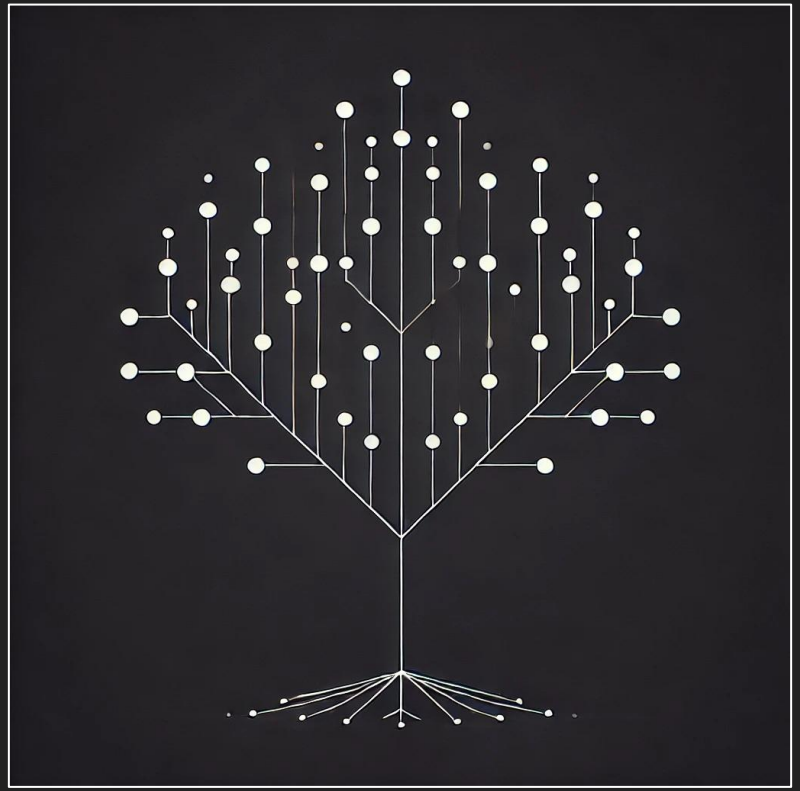
   Know exactly what you're doing - and how it works - theoretical side

2. Sound application in context

   Leverage (1) to win adversarial games - practical/technical aspects

Ex. You spent three months writing a custom coverage-guided fuzzer and it didn't find any bugs in time budget. Meanwhile, your friend wrote a "dumb" specialized fuzzer in a day and found a bug to win the contest in an hour. Who fuzzed smarter?

Part I. First Principles of Fuzzing



Enter Probability Theory

Probability & ops

Given a random variable x:

$p(x) \in [0..1]$: $x = x$

Joint probability for independent variables:

$\forall\ x \in x, y \in y$: $p(y=y, x=x) = p(x=x)p(y=y)$

Conditional probabilities:

$p(y=y|x=x) = p(y=y, x=x) / p(x=x)$

Chain rule:

$p(x_1,x_2,\ldots,x_n) = p(x_1)\prod^{n}_{i=2}p(x_n|x_1,x_2,\ldots,x_{n-1})$

Probability Distribution

PMF:

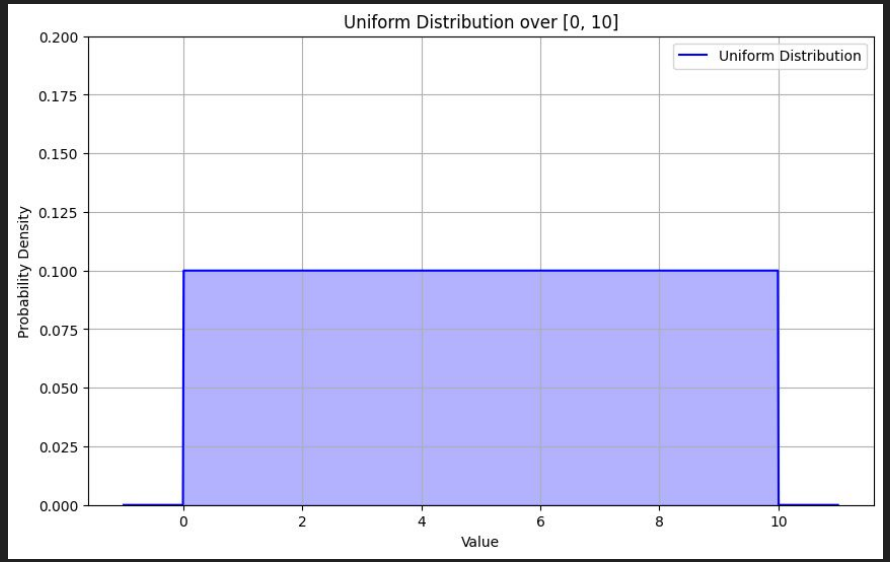
1. Dom(P): $\{x:x_i\}$, $i \in k$
2. $\forall x \in x$, $0 \leq P(x) \leq 1$
3. $\sum P(x) = 1^*$

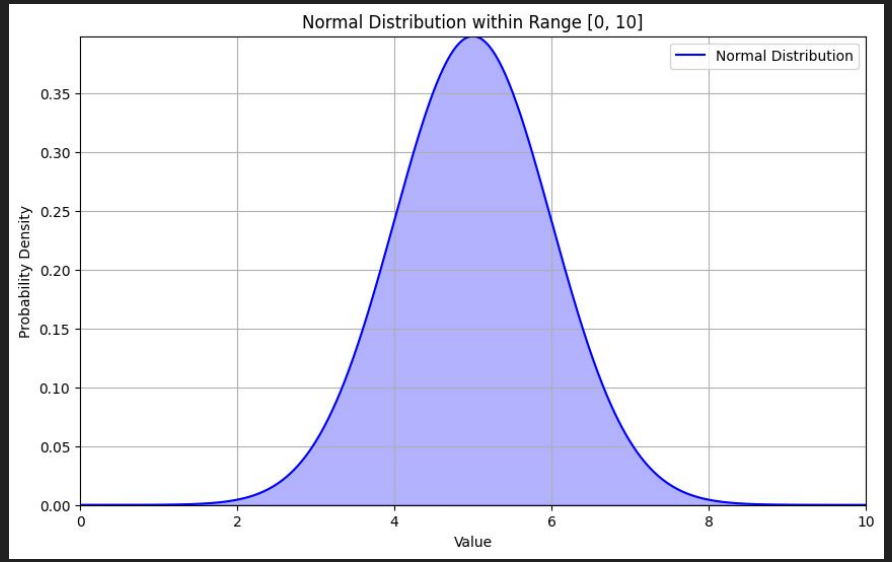
Ex. Uniform PD:

$p(x=x_i) = 1/k$

* normalised

Examples of common probability distributions

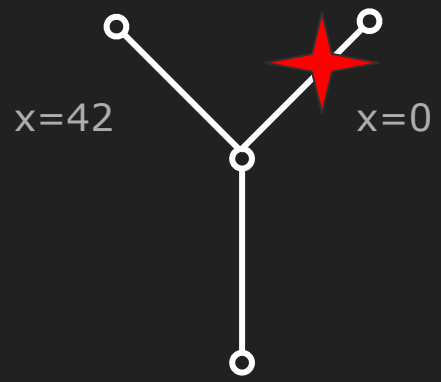




Modeling fuzzing of a simple program

Program:

1. *Accept one byte as input: x*
2. *If x = 42, print a message*
3. *Else if x = 0, proceed to vuln*
4. *Other inputs are invalid*
5. *Random fuzzing perspective*



Input vector

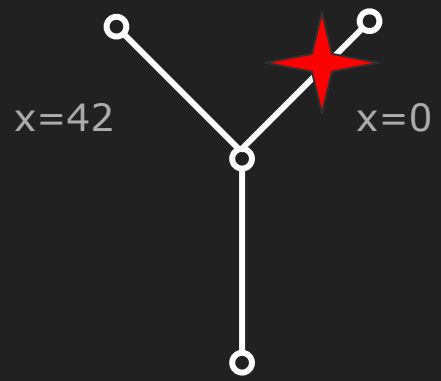
Then:

1. **Probabilities:**
   - p(x=42) = 1/256 = 0.039 (**3.9%**)
   - p(x=0) = 1/256 = 0.039 (3.9%)
2. **Success rate:**
   - 255/2 failures on average
   - 1/127 ~ 0.0079 (0.79%)
3. **Error rate:**
   - r ~ 1-0.0079 ~ 0.9921 (**99.21%**)
4. **Timing:** 100 cps ~> 1 sec to vuln

Vs. program's perspective

Program:

1. *Accept one byte as input: x*
2. *If x = 42, print a message*
3. *Else if x = 0, proceed to vuln*
4. *Other inputs are invalid*
5. *Program's perspective*



Input vector

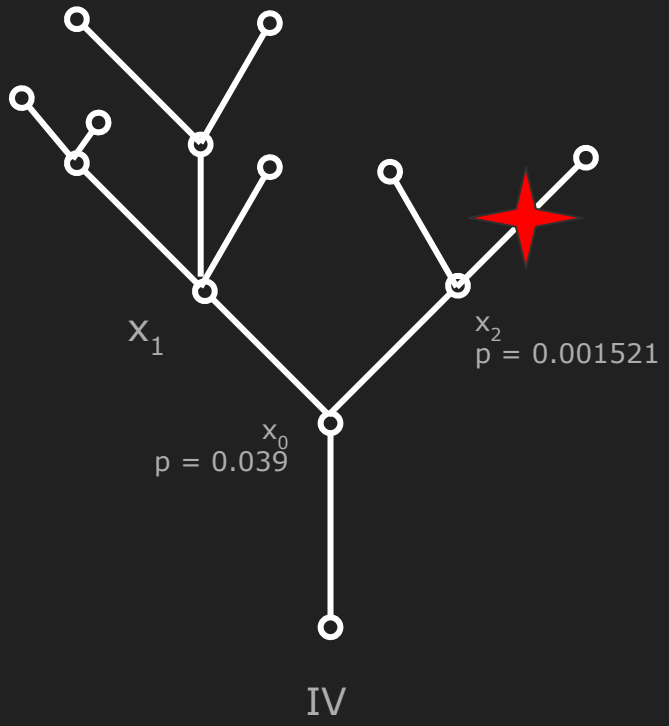
Then:

1. **Probabilities:**
   - p(x=42) = 0.5 (50%)
   - p(x=0) = 0.5 (50%)
2. **Success rate:**
   - 100% (not applicable)
3. **Error rate:**
   - 0 (not applicable)

Modeling a more complex program

Program:

1. *Accept an array of 10 bytes*
2. *Single bytes are tested*
3. *If x[0] = 42 AND x[2] = 0, proceed to vuln*
4. *Random fuzzing of each byte*



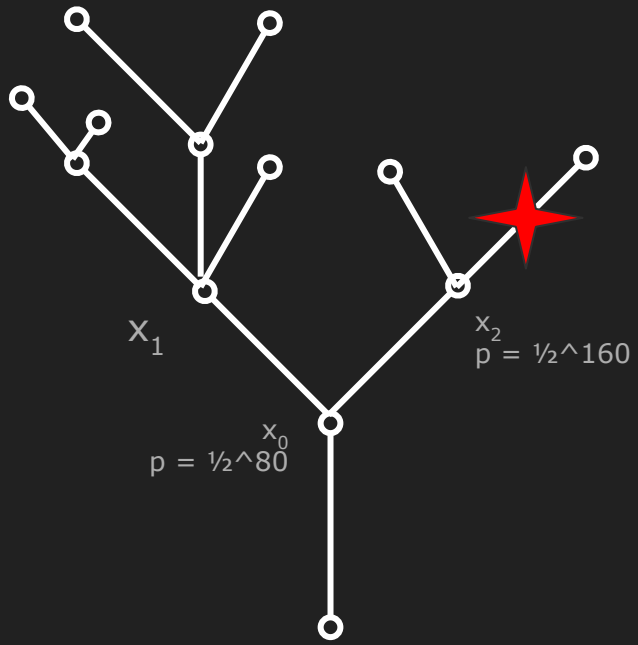
Then:

1. **Probabilities:**
   - $p(x_1=42, x_2=0) = 1/256*1/256 = 0.001521$ (**0.15%**)
   - **compounding**
2. Error rate
   - r ~ **99.85%**
   - Keeps growing up the tree
3. Overall trend
   - **exponential loss** due to path explosion + diminishing probabilities

Analytic: Behavior of a simple random fuzzer

Program fuzzing:

1. *Accept an array of 10 bytes*
2. *Random fuzzing of the array*
3. *If x[0] = 42 AND x[2] = 0, proceed to vuln*
4. *Else done*



IV

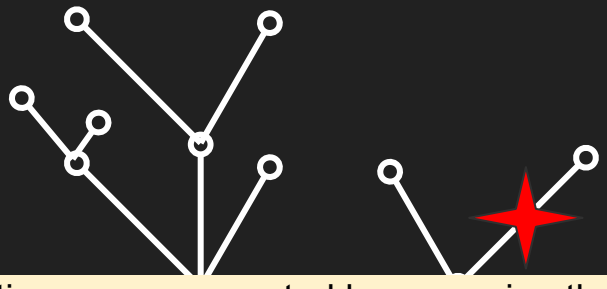
Then:

1. Tiny probability on each branch
   - $p(x_1=42) \sim 1/256^{10} \sim$ **$1/2^{80}$**
2. Compounding:
   - $p(x_1=42, x_2=0) \sim 1/2^{80} * 1/2^{80} \sim$ **$1/2^{160}$**
3. **Probability of reaching deeper branches with a fuzzer rapidly drops to 0**

Analytic: Behavior of a simple random fuzzer

Program fuzzing:

1. *Accept an array of 10 bytes*
2. *Random fuzzing*
3. *If (array == "FUZZP") proceed to Vuln*
4. *Else done*



$p = 1/2^{80}$

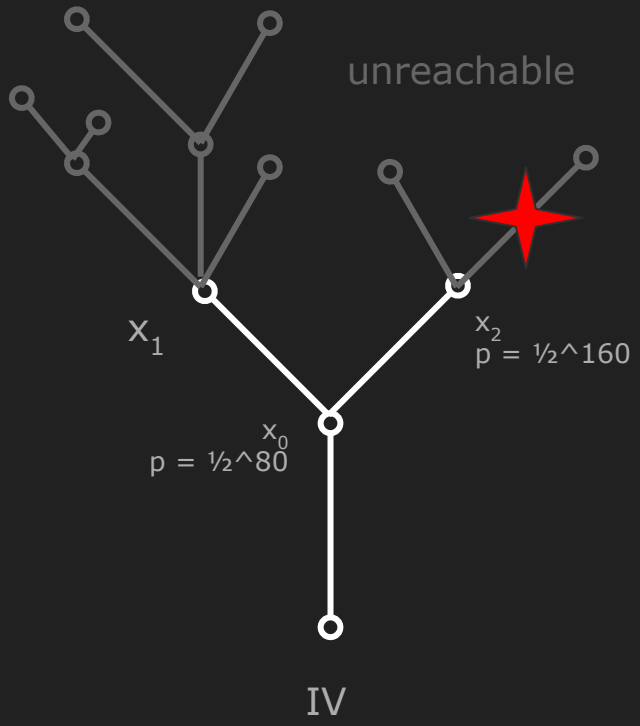
Then:

1. Tiny probability on each branch
   - $p(x_1=42) \sim 1/256^{10}$
   - $1/2^{80}$
2. Random fuzzing:
   - $p(x_1=0) \sim 1/2^{80} \sim 1/2^{80}$
3. **Probability of reaching deeper branches with a fuzzer rapidly drops to 0**

Note: Here, probabilities are exaggerated by assuming the worst case scenario, that entire 80-bit array is randomly fuzzed and tested as one big number. In practice, probabilities will depend on a number of variables: specific fuzzing algorithm, width of values tested on branches, offset of each value into the array, and so on.

My goal with this is to show two key points:
1) in real life non-deterministic fuzzing the probabilities of advancing into the program tree are terrible, and
2) just how much the probabilities can vary from the “ideal model”, depending on the specific fuzzing algorithm and other factors; and therefore, how steep the exponential path explosion curve can possibly get with an arbitrary choice of fuzzing techniques.

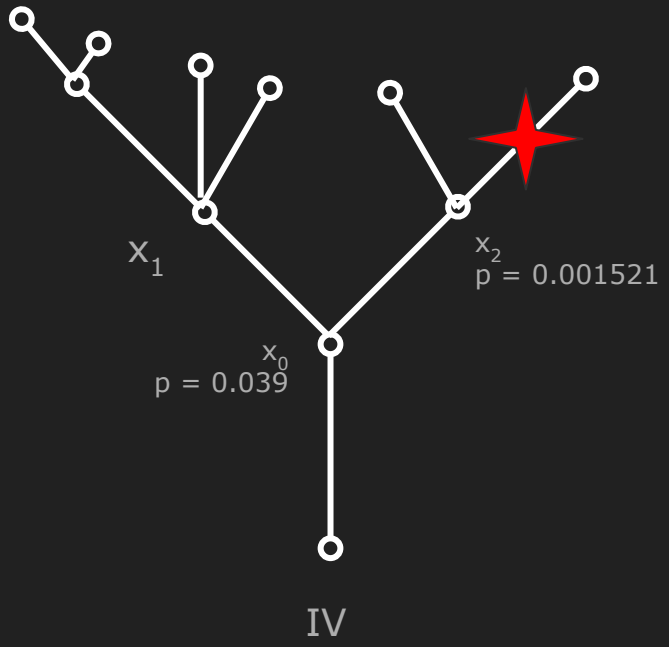
Analytic: Behavior of a simple random fuzzer



3. **Probability of reaching deeper branches with a fuzzer rapidly drops to 0**

Analytic: Behavior of a cov guided fuzzer 1 - branch discovery phase

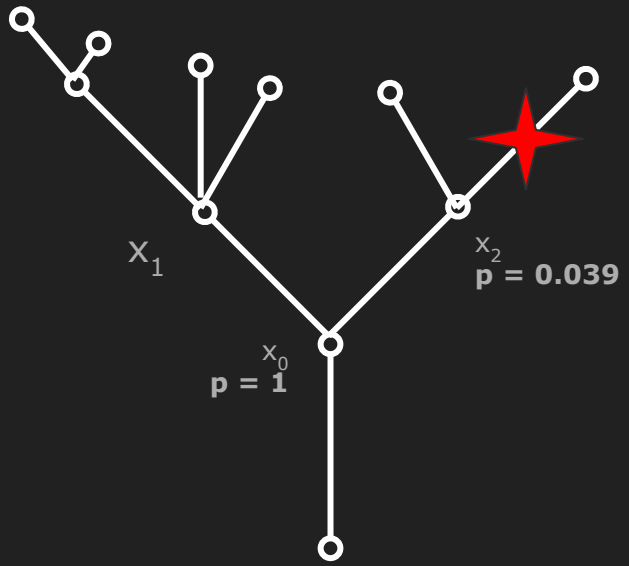
Assume that fuzzer is smart enough to fuzz specific byte that controls the branch so we don't get diminishing probabilities from fuzzing a very large number (not a real example - currently available SOTA fuzzers can't do it)



Then:

1. **Still low probability of discovering** each branch
2. Still 99%+ error rate **and growing through the tree**
3. Saving successful branch-passing inputs allows to **control the path explosion and overall exponential loss**

Analytic: Behavior of a cov guided fuzzer 2 - known branch fuzzing phase



Then:

1. Probabilities are improved!
   - Saving a sample sets **p=1 on the branch**
   - Further probabilities compound slower
2. **Cost is ~linear** rather than exponential!
3. Stable trend of vuln discovery (assuming a uniform distribution of bugs)
4. **Still 99%+++ error rate**

“I wonder, what my smart fuzzer is doing…”

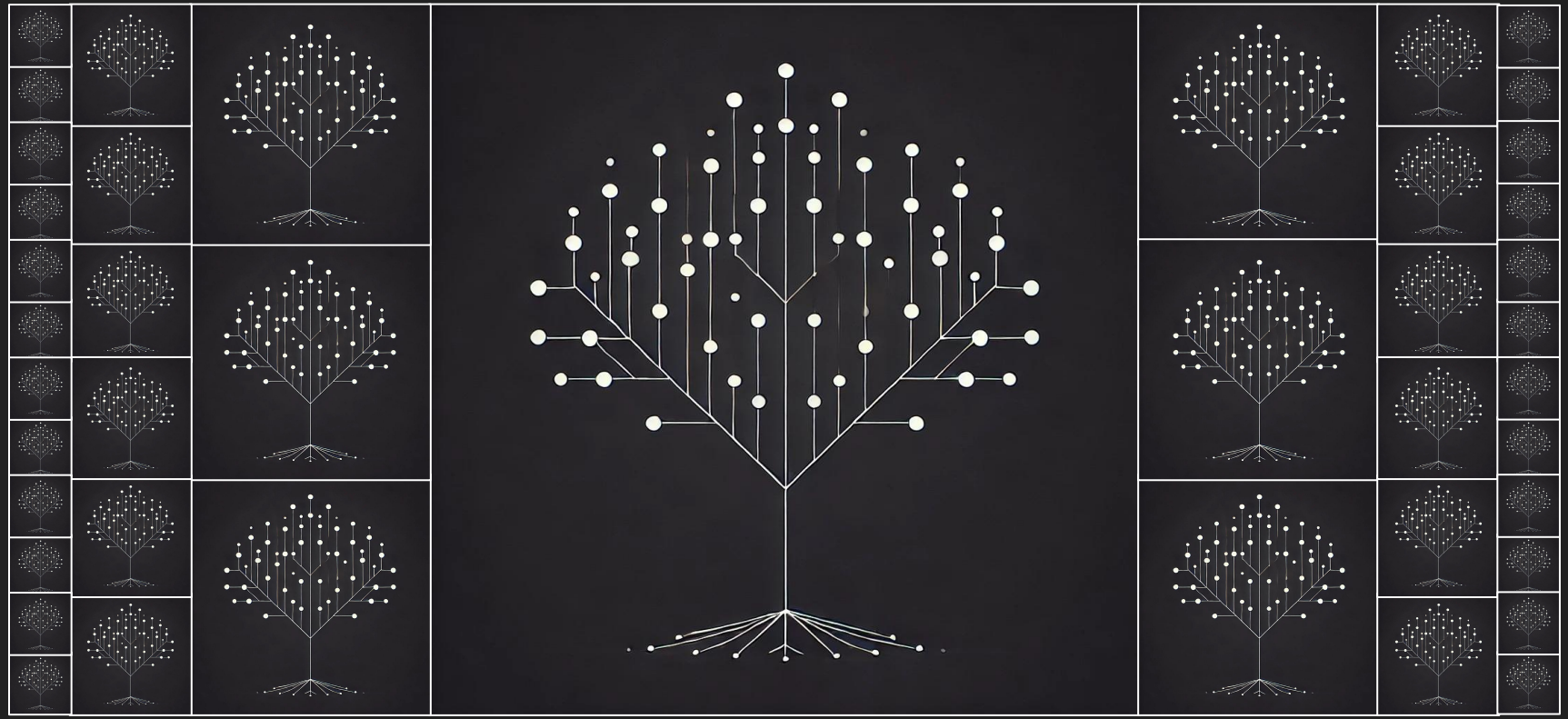
Popular opinion

“Walking in the forest of program paths, intelligently discovering new branches ☺️”

Reality

Burning electricity while producing heat 99+% of the time 🤔

Part II. The Forest of Probabilities



Fuzzing actors as Probability Distributions

Fuzzer

P_f

1. Over time a fuzzer will generate a finite* set of discrete values
2. Algorithm imposes constraints on possible values of values and their probabilities
3. Fuzzer = PD:

* limited by program’s input width

Fuzzing actors as Probability Distributions

Fuzzer

P_f

1. Over time a fuzzer will generate a finite* set of discrete values
2. Algorithm imposes constraints on possible values of values and their probabilities
3. Fuzzer = PD:

Program

P_p

1. Set of valid inputs
2. Valid input: pass at least one conditional branch
3. Algorithm imposes constraints on possible values of input and their probabilities
4. Program = PD:

* limited by program's input width

Fuzzing actors as Probability Distributions

Fuzzer

P_f

1. Over time a fuzzer will generate a finite* set of discrete values
2. Algorithm imposes constraints on possible values of values and their probabilities
3. Fuzzer = PD:

Program

P_p

1. Set of valid inputs
2. Valid input: pass at least one conditional branch
3. Algorithm imposes constraints on possible values of input and their probabilities
4. Program = PD:

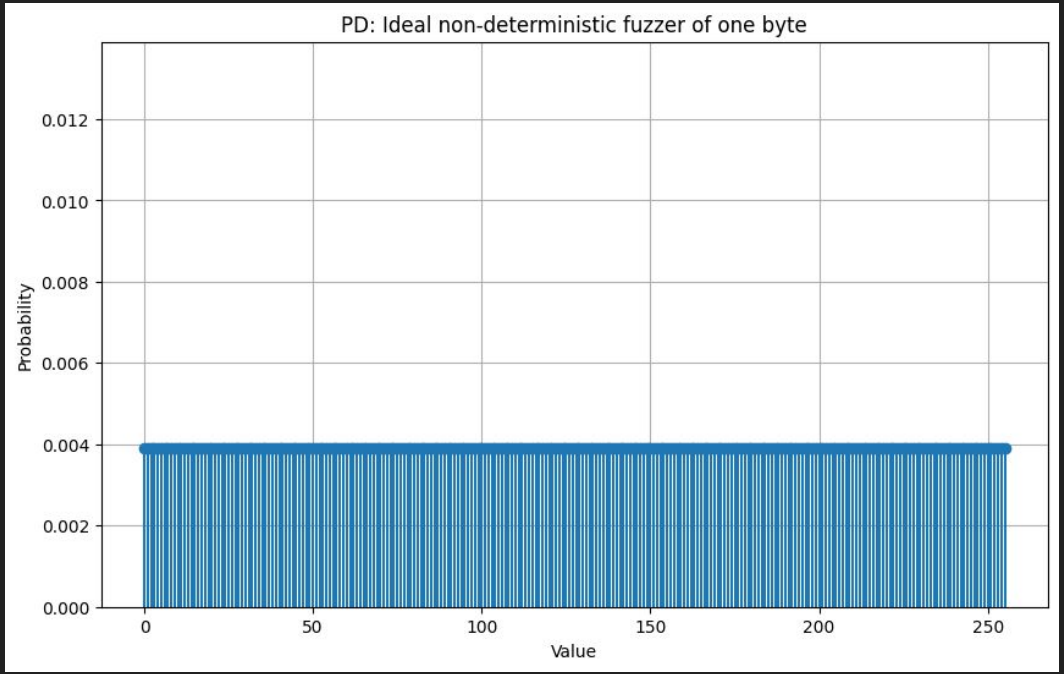
Vuln.

P_v

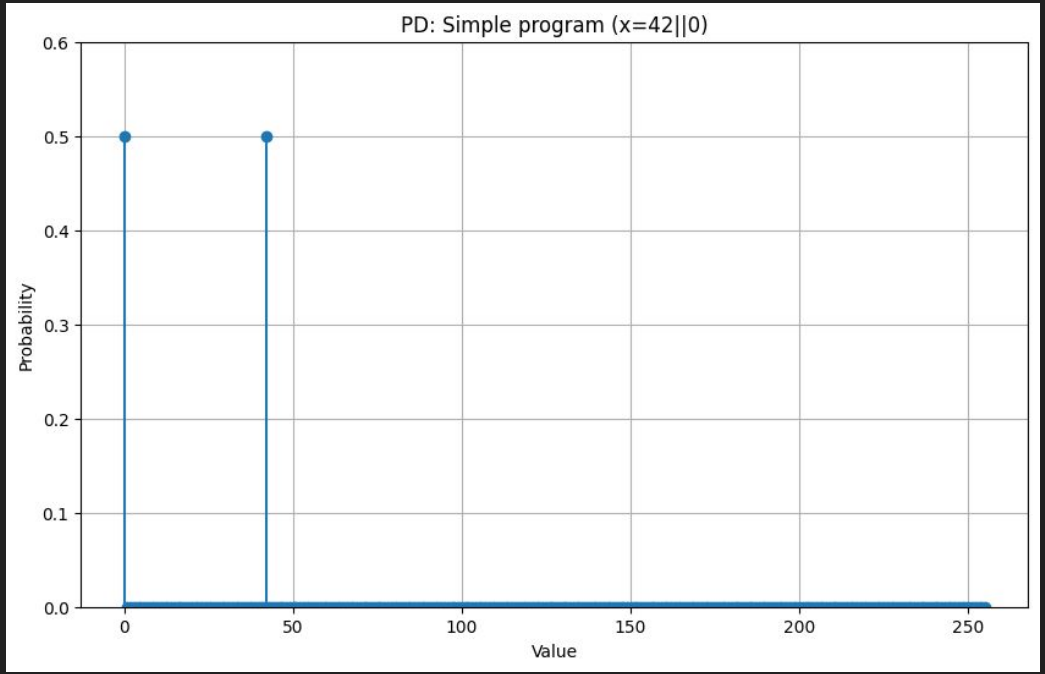
1. A strict subset of P_p
2. Specific values of program inputs that lead to vulnerabilities
3. PD:

* limited by program's input width

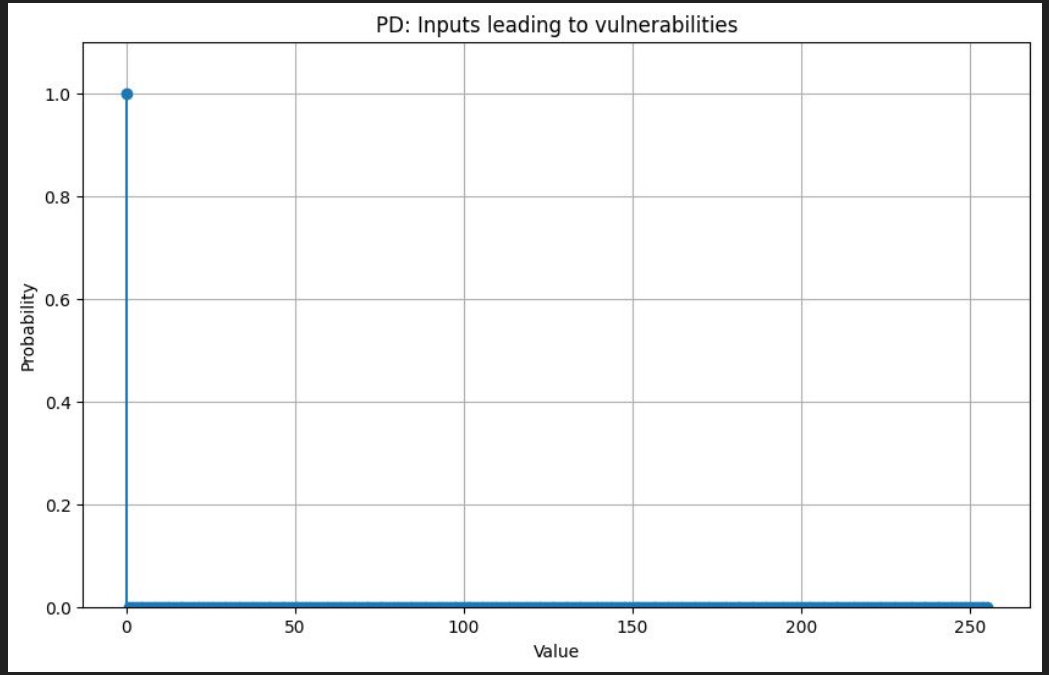
P_f


PD: Ideal non-deterministic fuzzer of one byte
Probability
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
50
100
150
200
250
Value

P_p


PD: Simple program (x=42||0)
Probability
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
50
100
150
200
250
Value

P_v


PD: Inputs leading to vulnerabilities
Probability
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
Value

Fundamental Challenge of Fuzzing

Goal: $P_f = P_v$

SOTA: $P_f \not\supset P_p \supset P_v$

Secondary goal: r = 0

(follows from $P_f = P_v$)

SOTA: r = 99%+↑

Modeling fuzzing as a process (rough idea)

Fuzzing is a function:

$$R = F(P_f)$$

R: practical measure of “success” of specific input values

Optimization:

$$R' = Opt(R, P_p) \rightarrow Opt(R, P_b) \rightarrow Opt(R, P_v)$$

$$P_f = F^{-1}(R')$$

Solve Fuzzing ($P_f = P_v$)*

1. $P_f \rightarrow P_p$

Idea: bring the set of fuzzer inputs closer to the set valid program inputs

Insight: fuzzer's losses are exponential in discovery phase and linear past it. Here we aim for latter stage upfront

* is a hard problem, so we also look for less ambitious and more trackable solutions in practice.

The probabilistic model has many solutions. Here as an example, I show four "partial" solutions that can be trivially illustrated with my own past fuzzing experiments.

Solve Fuzzing ($P_f = P_v$)*

1. $P_f \rightarrow P_p$

Idea: bring the set of fuzzer inputs closer to the set valid program inputs

Insight: fuzzer's losses are exponential in discovery phase and linear past it. Here we aim for latter stage upfront

2. $P_p \rightarrow 0$

Idea: minimize the set of valid program inputs

Insight: Random fuzzing with small P_p is more efficient than smart coverage-guided fuzzing of the entire program (todo: proof)

* is a hard problem, so we also look for less ambitious and more trackable solutions in practice.

The probabilistic model has many solutions. Here as an example, I show four "partial" solutions that can be trivially illustrated with my own past fuzzing experiments.

Solve Fuzzing ($P_f = P_v$)*

1. $P_f \rightarrow P_p$

Idea: bring the set of fuzzer inputs closer to the set valid program inputs

Insight: fuzzer's losses are exponential in discovery phase and linear past it. Here we aim for latter stage upfront

2. $P_p \rightarrow 0$

Idea: minimize the set of valid program inputs

Insight: Random fuzzing with small P_p is more efficient than smart coverage-guided fuzzing of the entire program (todo: proof)

3. $P_f \rightarrow P_v$

Idea: let the fuzzer "learn" from past bugs

Insight: this always "tends" with a gap: eg, next bug is a novel one - unseen previously. The gap is an opportunity in itself

* is a hard problem, so we also look for less ambitious and more trackable solutions in practice.

The probabilistic model has many solutions. Here as an example, I show four "partial" solutions that can be trivially illustrated with my own past fuzzing experiments.

Solve Fuzzing ($P_f = P_v$)*

1. $P_f \rightarrow P_p$

Idea: bring the set of fuzzer inputs closer to the set valid program inputs

Insight: fuzzer's losses are exponential in discovery phase and linear past it. Here we aim for latter stage upfront

2. $P_p \rightarrow 0$

Idea: minimize the set of valid program inputs

Insight: Random fuzzing with small P_p is more efficient than smart coverage-guided fuzzing of the entire program (todo: proof)

3. $P_f \rightarrow P_v$

Idea: let the fuzzer "learn" from past bugs

Insight: this always "tends" with a gap: eg, next bug is a novel one - unseen previously. The gap is an opportunity in itself

4. $P_f \rightarrow 1$

Idea: Collapse it (theoretical)

Insight: manifestation ???

* is a hard problem, so we also look for less ambitious and more trackable solutions in practice.

The probabilistic model has many solutions. Here as an example, I show four "partial" solutions that can be trivially illustrated with my own past fuzzing experiments.

Reality check

~100% of software security bugs exploited at Pwn2Own Vancouver 2024 were found with manual program analysis

Meaning: for purposes of adversarial offensive security research, fuzzing is so bad that researchers don’t even try it?

Actually, use of scrappy custom **ultra specialized fuzzers** is common in these scenarios (optimizing P_f intuitively), a bug will look like a manual find in this case. But researchers normally won’t publish it because it isn’t considered “smart fuzzing”.

Case studies

Summary: concrete examples of F improvement

1. $P_f \rightarrow P_p$

Idea: bring the set of fuzzer inputs closer to the set valid program inputs

Examples:

- Mutate valid program inputs. (*radamsa*)
- Generate valid inputs with context free grammars. (*dharma*)
- Reuse existing parser code to gen F inputs.*
- Coverage feedback (*afl, winafl, libfuzzer*)

* Similar to genai, where the originally analytical model is reversed to produce outputs.

2. $P_p \rightarrow 0$

Idea: minimize the set of valid program inputs

Examples:

- Cut out a piece of code and fuzz it separately of the entire program.

- Hook into a the code to isolate fuzzing of a specific portion. (*frida*)

- Use one particular valid input as a fuzzing template to isolate a specific portion of the program tree as a target.

3. $P_f \rightarrow P_v$

Idea: let the fuzzer "learn" from past bugs

Examples:

- Take one of past bugs, reuse it as a fuzzing template.

Note, this is a crude 'fuzzer' that has a very small P_f ~ P_v for one specific vuln.

- Reuse the corpus of past bugs by feeding it into the fuzzer as input corpus.

4. $P_f \rightarrow 1$

Idea: Collapse it (theoretical solution)

Side note: much of this has been empirically discovered by the fuzzing community over many years of practical research, and implemented in various tools and best practices, which validates this theoretical model!

Time to start thinking forward from the model to guide new fuzzing improvements, rather than stumble on them empirically and intuitively.

Model applications: from basics to advanced

High level

Scale: Probability Distributions

Goal: $P_f \rightarrow P_v$

Ex.(Idea) Quantify the difference between PDs -> cost function -> optimization algorithm over fuzzing procedures or fuzzer code

Low level

Scale: Branch/instruction

Goal: max(p) | min(r)

Ex.(Idea) Find a way to make fuzzer aware of width and/or position of condition value on a branch => higher probability of finding the value => faster branch discovery + lower error rate

Conclusions

1. Fuzzing from First Principles enables us to compete with large scale “smart” fuzzing; it doesn’t have to be manual analysis
2. Coverage guided and scaled fuzzing is ok for long-running projects with little or no constraints - ex: part of dev cycle (but still very far from reasonable in current SOTA)
3. Cov guided and scaled fuzzing seems to be a bad choice in adversarial games of offensive security (0-days, competitions, bug bounties)
4. Improve P_f instead of glossing over the failure scaling madness
5. **Start thinking forward from the theoretical model to guide advancements in fuzzing, rather than stumble on improvements empirically and intuitively in practice.**

Discussion

contact@zerodayengineering.com

@alisaesage