Scheduling Black-box Mutational Fuzzing

ACM CCS 2013

Maverick Woo
Carnegie Mellon University
pooh@cmu.edu

Our Crew

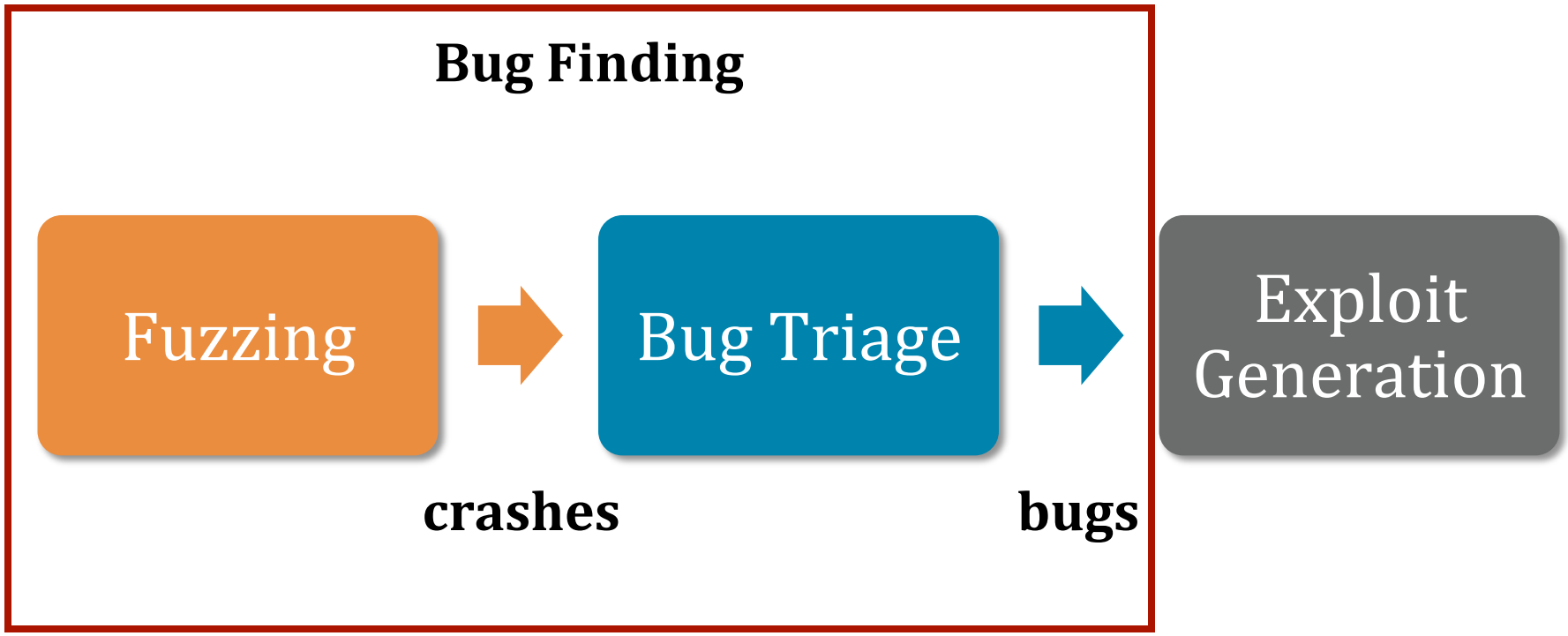
Maverick Woo **Sang Kil Cha** **Samantha Gottlieb** **David Brumley**

The Story

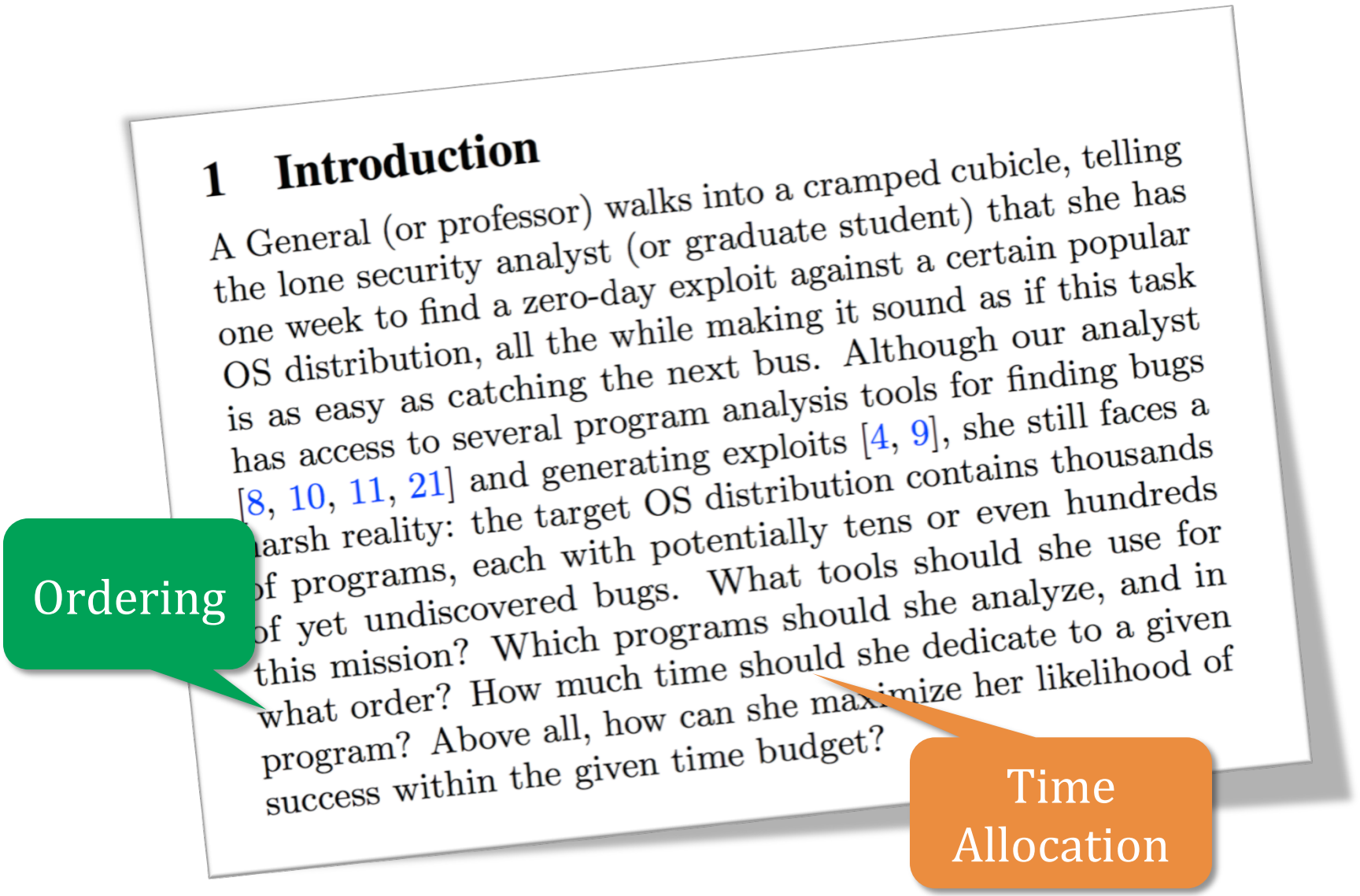
1 Introduction

A General (or professor) walks into a cramped cubicle, telling the lone security analyst (or graduate student) that she has one week to find a zero-day exploit against a certain popular OS distribution, all the while making it sound as if this task is as easy as catching the next bus. Although our analyst has access to several program analysis tools for finding bugs [8, 10, 11, 21] and generating exploits [4, 9], she still faces a harsh reality: the target OS distribution contains thousands of programs, each with potentially tens or even hundreds of yet undiscovered bugs. What tools should she use for this mission? Which programs should she analyze, and in what order? How much time should she dedicate to a given program? Above all, how can she maximize her likelihood of success within the given time budget?

Typical Exploit Generation



Scheduling is Equally Important



1 Introduction

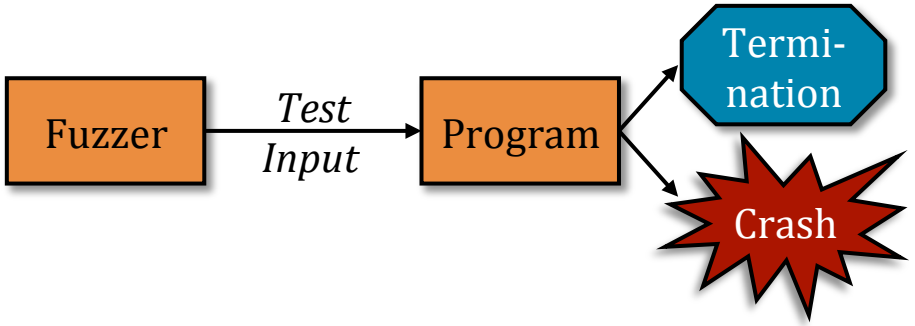
A General (or professor) walks into a cramped cubicle, telling the lone security analyst (or graduate student) that she has one week to find a zero-day exploit against a certain popular OS distribution, all the while making it sound as if this task is as easy as catching the next bus. Although our analyst has access to several program analysis tools for finding bugs [8, 10, 11, 21] and generating exploits [4, 9], she still faces a harsh reality: the target OS distribution contains thousands of programs, each with potentially tens or even hundreds of yet undiscovered bugs. What tools should she use for this mission? Which programs should she analyze, and in what order? How much time should she dedicate to a given program? Above all, how can she maximize her likelihood of success within the given time budget?

Ordering

Time Allocation

Scheduling Black-box Mutational Fuzzing

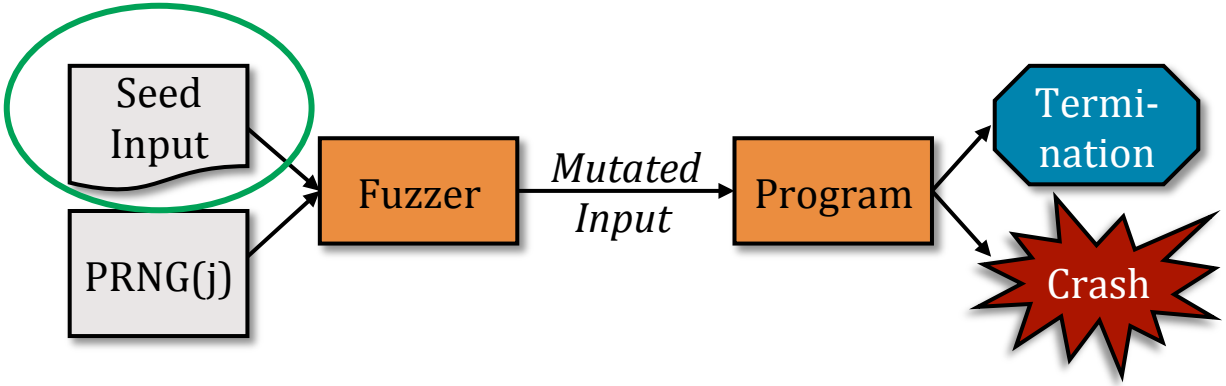
Scheduling Black-box Mutational Fuzzing



A common program testing technique popularized by Miller et al. in late 1980s [18]

- Use a ***fuzzer*** to generate ***test inputs*** to program-under-test
- At its simplest, look for ***crashes***—memory corruption, uncaught exceptions, failed assertions, etc.

Scheduling Black-box Mutational Fuzzing

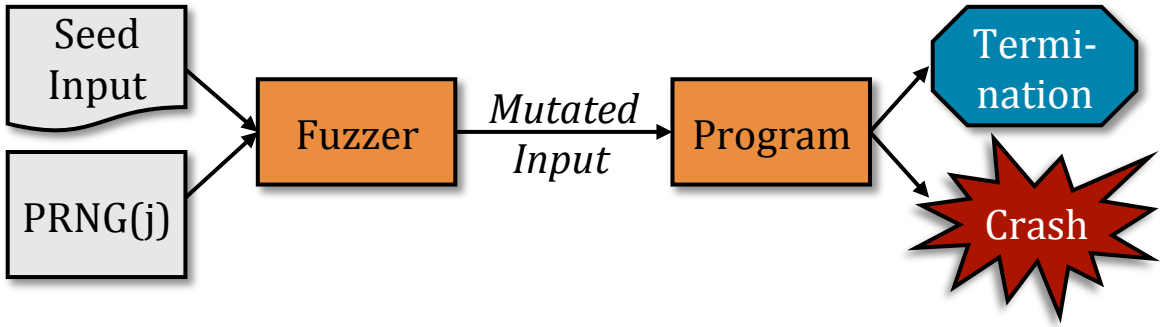


A ***black-box*** fuzzer observes a program's ***I/O behavior*** only

- cf. Whitebox Fuzzing by Godefroid et al. 2012 [11]
- Simplification: only distinguish ***termination*** vs. ***crash***

Detect anomaly by *mutating* a valid input (= seed)

Scheduling Black-box Mutational Fuzzing



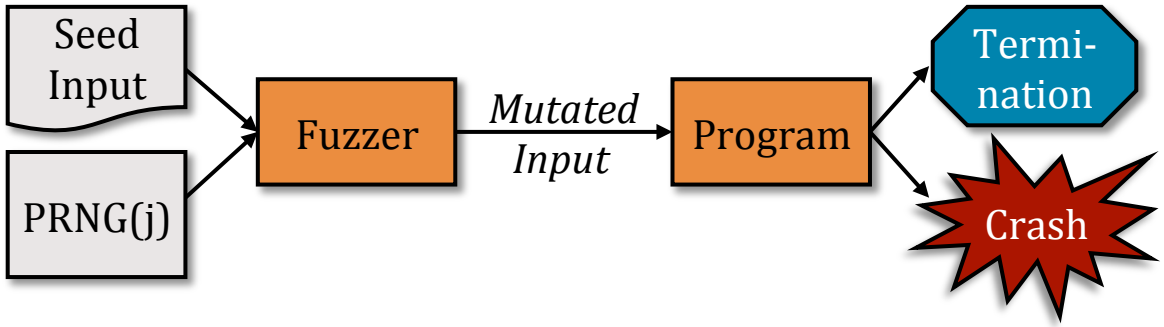
A ***black-box*** fuzzer observes a program's ***I/O behavior*** only

- cf. Whitebox Fuzzing by Godefroid et al. 2012 [11]
- Simplification: only distinguish ***termination*** vs. ***crash***

Given a ***seed input*** s and a ***mutation ratio*** r:

1. Select $d = r \times |s|$ bits in s uniformly at random
2. Flip each selected bit with probability ½

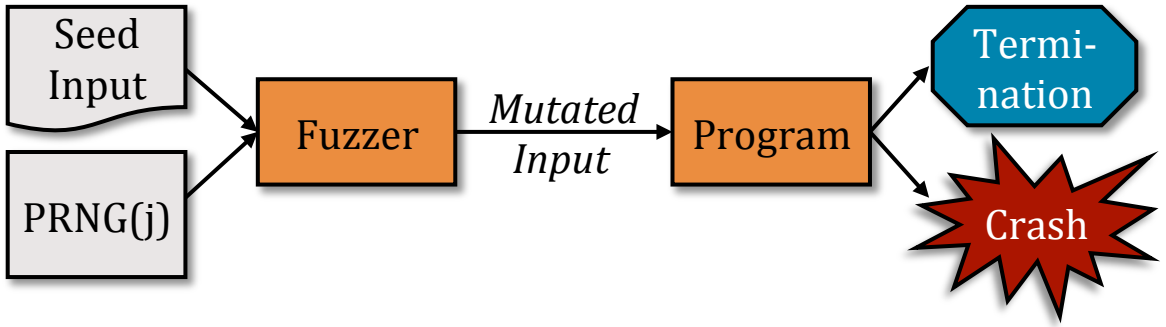
Scheduling Black-box Mutational Fuzzing



Key Observations:

1. We can ***reproduce*** a program crash by storing (a) the seed input and (b) the PRNG seed
2. Mutation = ***uniform sampling*** from the Hamming cube of radius d centered at s

Scheduling Black-box Mutational Fuzzing



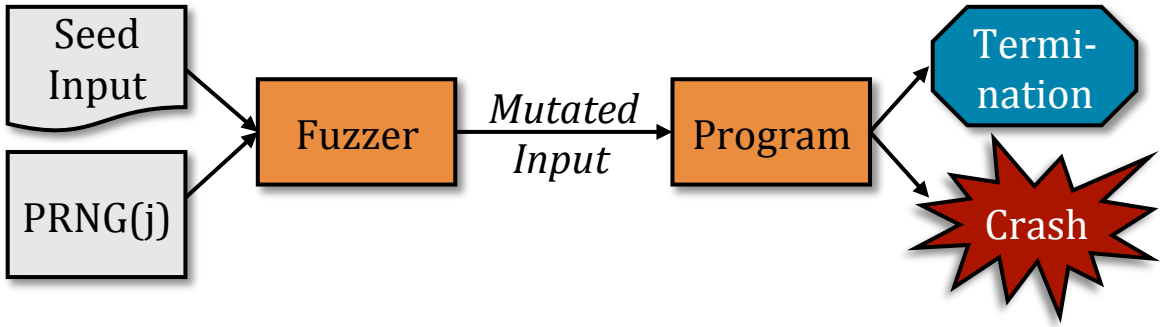
Key Observations:

1. We can ***reproduce*** a program c
   (a) the seed input and (b) the P
2. Mutation = ***uniform sampling***
   radius *d* centered at *s*

"Fuzz Configuration"

(i) program *p*
(ii) seed input *s*
(iii) mutation ratio *r*

Scheduling Black-box Mutational Fuzzing



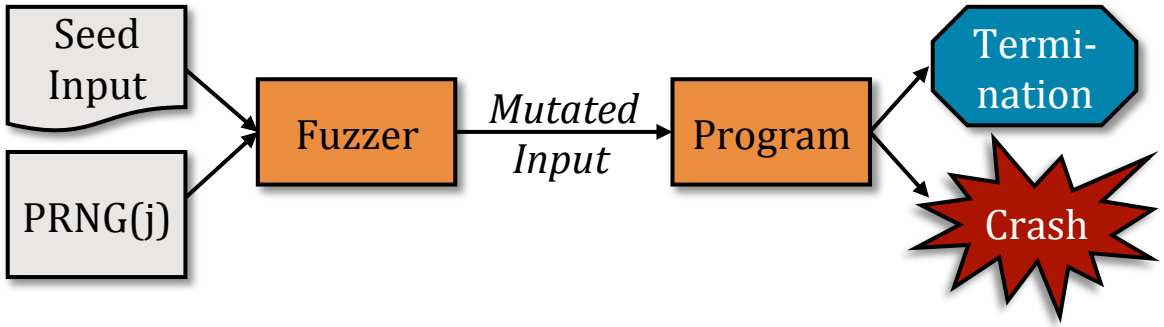
Key Observations:

1. We can ***reproduce*** a program c
   (a) the seed input and (b) the P
2. Mutation = ***uniform sampling***
   radius *d* centered at *s*

"Fuzz Configuration"

(i) program *p*
(ii) seed input *s*
(iii) 0.04%

Scheduling Black-box Mutational Fuzzing



Key Observations:

1. We can ***reproduce*** a program crash by storing (a) the seed input and (b) the P
2. Mutation = ***uniform sampling*** radius d centered at s

"Fuzz Configuration"
=
"(program, seed) pair"
in this talk

Scheduling Black-box Mutational Fuzzing

A ***fuzz campaign*** comprises a sequence of ***epochs***:

1. takes a list of (program, seed) pairs as input
2. at the beginning of each epoch, picks ***one*** (program, seed) pair to fuzz based on data collected from previous epochs

We investigate two ***epoch types***:

- **Fixed-run**: fixed number of fuzz runs in each epoch
  - implemented in CMU CERT BFF v2.6 [14]
- **Fixed-time**: fixed amount of time in each epoch
  - proposed in this paper
  - slightly harder to implement

Problem Statement

Given a list of K fuzz configurations $\{(p_1, s_1), \cdots, (p_K, s_K)\}$, the ***Fuzz Configuration Scheduling (FCS)*** problem seeks to maximize the number of *unique* bugs discovered in a fuzz campaign that runs for a duration of length T.

Important Assumptions:

1. Only ***one*** configuration can be fuzzed within an epoch
2. Separate program analysis of (p_i, s_i) is ***not*** allowed
3. Bugs from different (p_i, s_i) are ***disjoint***

See paper for discussions

How to Solve the FCS Problem?

Two ***competing goals*** during a fuzz campaign:

Explore each (p_i, s_i) sufficiently often so as to identify pairs that can yield new bugs

Exploit knowledge of (p_i, s_i) that are likely to yield new bugs by fuzzing them more

Good News:

- Clearly a ***Multi-Armed Bandit (MAB)*** problem!

Multi-Armed Bandits

MAB in Berlin

How to Solve the FCS Problem?

Two ***competing goals*** during a fuzz campaign:

Explore each (p_i, s_i) sufficiently often so as to identify pairs that can yield new bugs

vs.

Exploit knowledge of (p_i, s_i) that are likely to yield new bugs by fuzzing them more

Good News:

- Clearly a ***Multi-Armed Bandit (MAB)*** problem!
- ***Lots*** of published MAB algorithms
  - ***provably optimal*** algorithms for many settings, e.g., Auer et al. 2002 [2] handles certain ***adversarial*** cases

How to Solve the FCS Problem?

Bad News: recognizing "FCS ∈ MAB" is **not** enough

Given a list of K fuzz configurations $\{(p_1, s_1), \cdots, (p_K, s_K)\}$, the ***Fuzz Configuration Scheduling (FCS)*** problem seeks to maximize the number of *unique* bugs discovered in a fuzz campaign that runs for a duration of length T.

1. Classic MAB: once you identify a good beer, it ***stays*** good
   ⇒ drink it often to ***accumulate*** rewards ☺
2. Our Setting: each program has a ***finite*** number of bugs
   ⇒ bug ***exhaustion*** gives a diminish of return ☹

We are not aware of MAB algorithms that cater to our case…
⇒ **We need our own algorithms!**

How to Solve the FCS Problem?

Bad News: recognizing "FCS ∈ MAB" is **not** enough

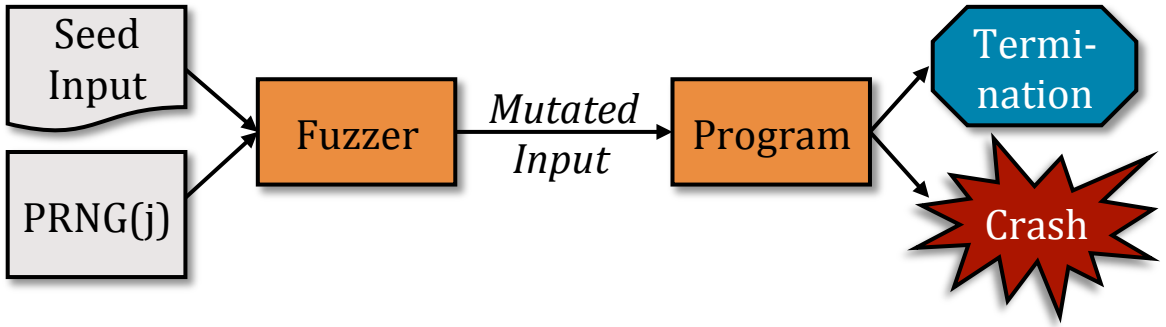
Given a list of K fuzz configurations $\{(p_1, s_1), \cdots, (p_K, s_K)\}$, the ***Fuzz Configuration Scheduling (FCS)*** problem seeks to maximize the number of *unique* bugs discovered in a fuzz campaign that runs for a duration of length T.

1. Classic MAB: once you identify a good beer, it ***stay***od
   ⇒ drink it often to ***accumulate*** rewards ☺
2. Our Setting: each program has a ***finite***
   ⇒ bug ***exhaustion*** gives a dimin

We are not aware of MAB algorithm…ase…
⇒ **We need our own algorithms!**

Previously

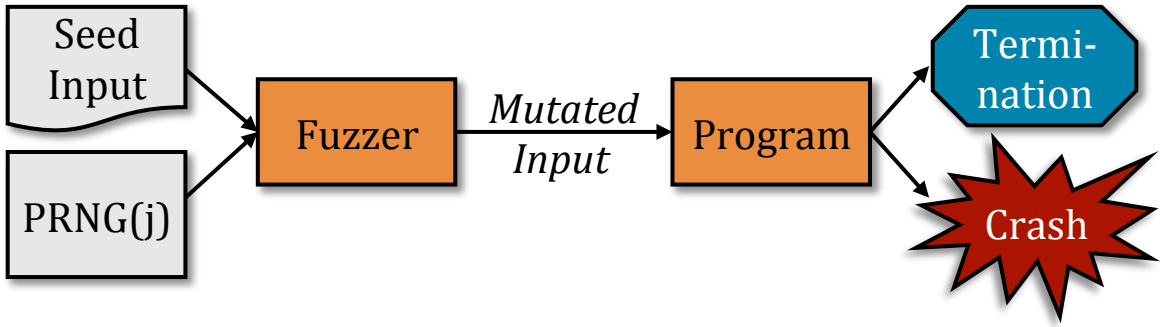
Scheduling Black-box Mutational Fuzzing



Key Observations:

1. We can ***reproduce*** a program crash by storing (a) the seed input and (b) the PRNG seed
2. Mutation = ***uniform sampling*** from the Hamming cube of radius d centered at s

Modeling Black-box Mutational Fuzzing



Consider the repeated fuzzings of a ***fixed*** (p_i, s_i) and let $outcome_i(j)$ denote the j-th outcome in the sequence:

- Termination ⇒ ID 0
- Crash ⇒ bug ID obtained from bug triage

Key Observation:

BMF is ***memoryless***, i.e., $outcome_i(j)$ are ***i.i.d.*** RVs for a fixed i

Coupon Collector's Problem (CCP)

Suppose every box of breakfast cereal comes with a coupon that is randomly chosen among M different coupon types

- How many boxes do you ***expect*** to buy before you have collected ***at least one*** coupon of each type?

Traditional Setting

- Coupon types are uniformly distributed $\Rightarrow \Theta(M \log M)$

Our Setting

- Bugs do not occur uniformly at random $\Rightarrow$ ***Weighted*** **CCP**
- Prevalence of different bugs is ***unknown ahead of time***

Coupon Collector's Problem (CCP)

Suppose every box of breakfast cereal comes with a coupon that is randomly chosen among M different coupon types

- How many boxes do you ***expect*** to buy before you have collected ***at least one*** coupon of each type?

Traditional Setting

- Coupon types are uniformly distributed $\Rightarrow \Theta(M \log M)$

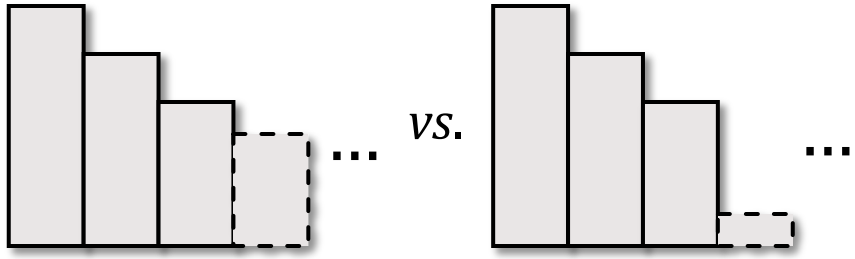
Our Setting

- Bugs do not occur uniformly at random =
- Prevalence of different bugs is ***unknown***

Also observed by Arcuri 2010 [1]

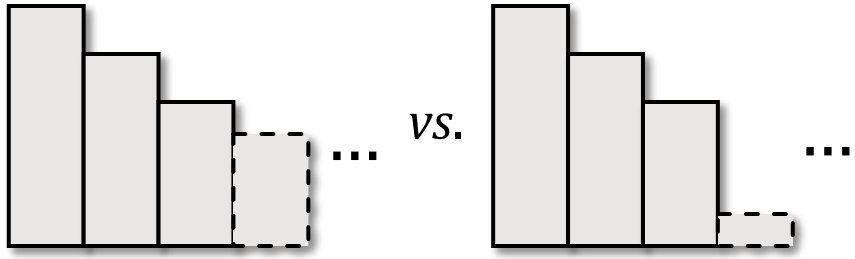
WCCP w/ Unknown is Intractable

No Free Lunch Theorem
(you did pay the registration, *right?*)



WCCP w/ Unknown is Intractable

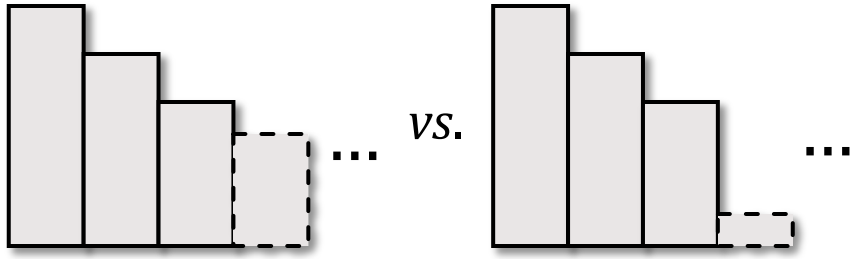
No Free Lunch Theorem
Wolpert and Macready 2005 on [22]



- "Any two optimization algorithms are ***equivalent*** when their performance is averaged across ***all*** possible problems"

"Bring Your Own Prior"

No Free Lunch Theorem
Wolpert and Macready 2005 on [22]



- "Any two optimization algorithms are ***equivalent*** when their performance is averaged across ***all*** possible problems"

Circumvention may be possible!

- NFL Theorem does not apply if we focus on distributions that are ***more likely*** to occur in ***practice***
- More accurate ***model*** ⇒ More accurate ***predictions*** ⇒ More ***bugs***

Rule of Three

Q: Suppose we have flipped a biased H-T coin n times and every time it comes up H. Does Pr[T] ***have*** to be small?

A: No, so long as Pr[T] < 1, our observation is always ***possible***

Confidence Intervals:
Pr[T] < 3/n in 95% of all "parallel universes"

See discussion in Jovanovic 1997 [15]

Usage:

1. Suppose (p_i, s_i) has yielded n different outcomes so far
2. Collectively call all n outcome types H
3. With 95% confidence, Pr[T (i.e., new outcome)] < 3/n

Algorithm Design Space

We explore **3 *dimensions*** in algorithm design and present:

- **2 Epoch Types**
  - fixed-run
  - fixed-time
- **5 MAB Algorithms**
  - Round-Robin
  - Uniform-Random
  - EXP3.S.1 from Auer et al. 2002 [2]
  - Weighted-Random
  - ε-Greedy
- **5 Belief Metrics**

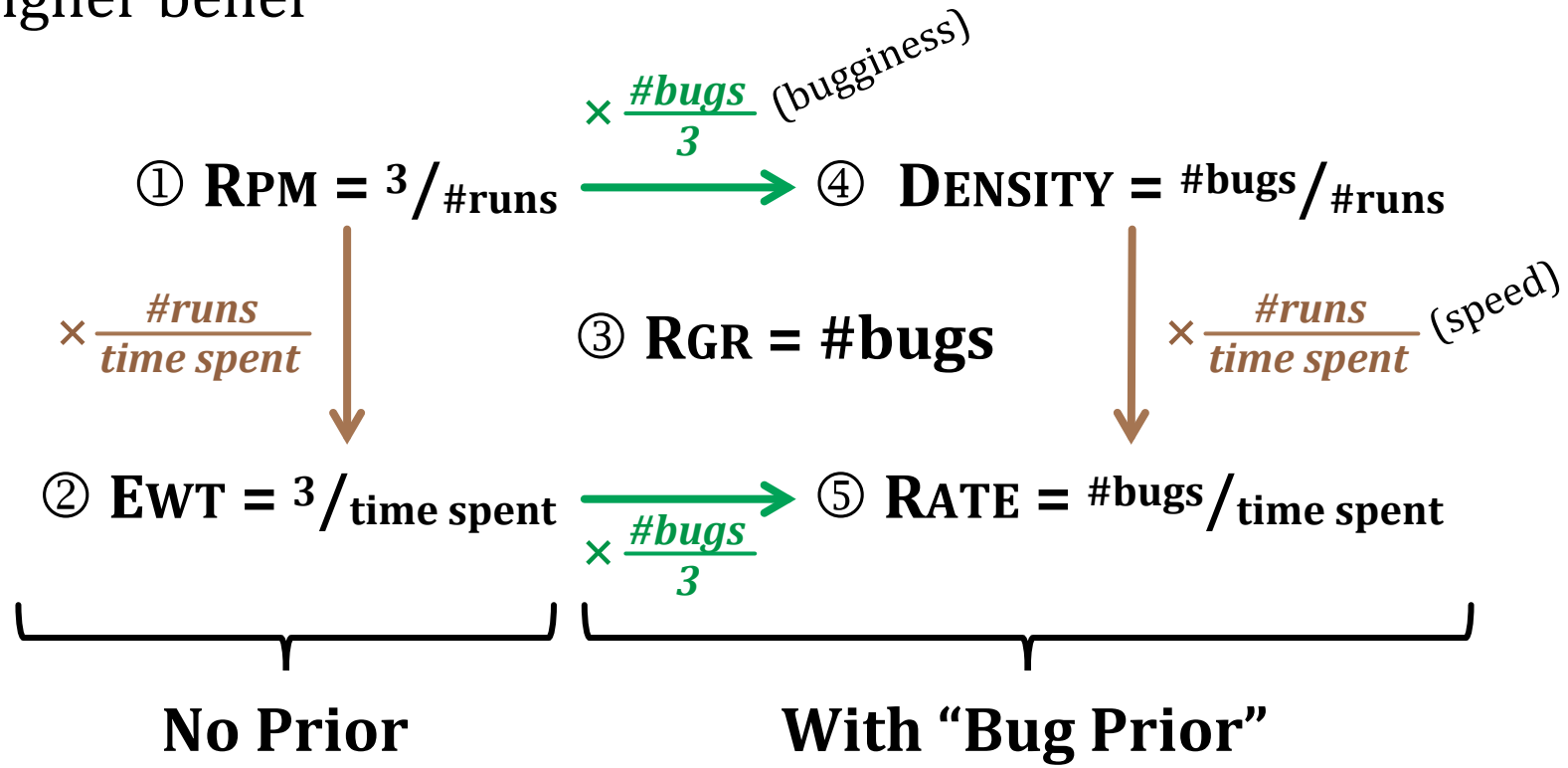
w.r.t. belief metrics

2 * (3 + 2 * 5) = 26 Scheduling Algorithms

Belief Metrics

The ***belief*** over (p_i, s_i) is a heuristic to estimate the likelihood of yielding a ***new*** outcome in the next fuzz run of this pair

- Weighted-Random & ε-Greedy both bias towards pairs with higher belief



The Evaluation Challenge

1. **Find *large* & *representative* data sets**

If an algorithm performs well on such data sets, then we gain confidence that it is superior for current practice

2. **How *good* is an algorithm, really?**

Is an algorithm that finds 200 bugs in 10 days ***good*** or ***bad***?
⇒ Need to ***know*** max #bugs that can be found in 10 days, but this is circular! We are trying to solve ***this*** problem!

3. **How to try many algorithms *affordably*?**

Yes, we tried ***way*** more than 26 combinations… ☺

How To Pull This Off

http://s3.amazonaws.com/rapgenius/filepicker%2FgkTHRLQsyzS3MggKloYA_money.jpg

How To Pull This Off

Step 1. Select two ***representative*** datasets:

Intra-Program: 100 randomly-sampled seeds for FFMPEG
Inter-Program: 100 file converters in Debian w/ valid seeds

Step 2. Fuzz ***each*** of the 200 pairs on EC2 for 10 days—

48,000 CPU hours (~5.5 CPU years) later:

Dataset	#runs	#crashes	#bugs
Intra-program	636,998,978	906,577	200
Inter-program	4,868,416,447	415,699	223

Table 1: Statistics from fuzzing the two datasets.

Step 3. Build the FUZZSIM ***replay*** system to ***simulate*** any scheduling algorithm with ***no*** additional fuzzings

FUZZSIM Overview

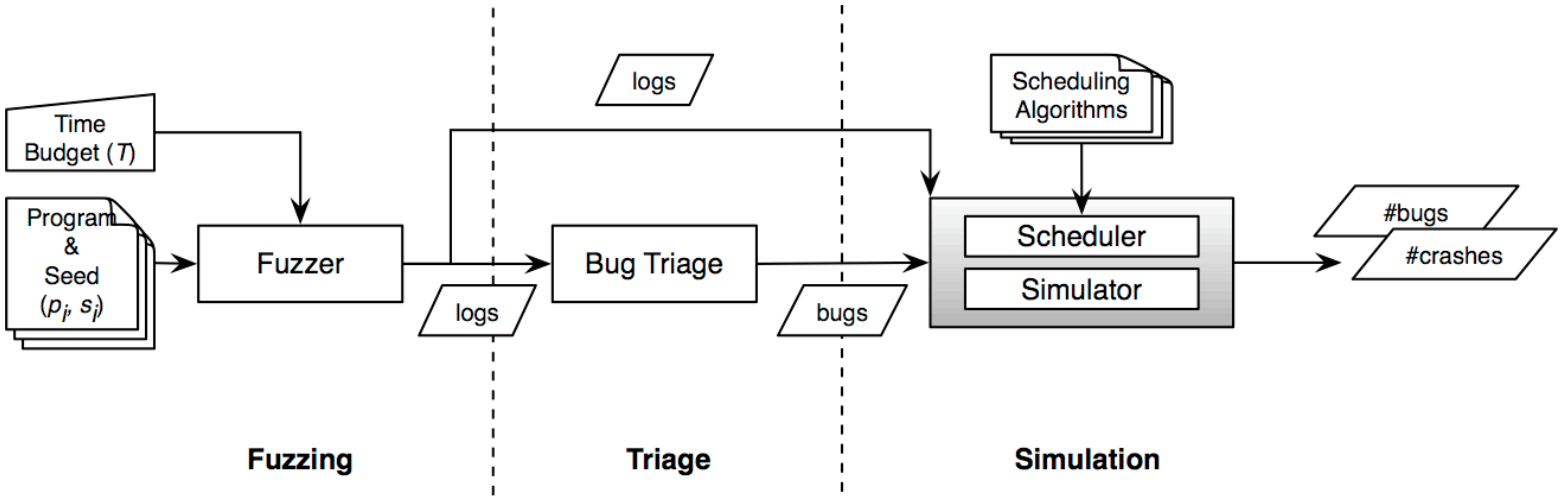


Figure 1: FUZZSIM architecture.

- **Example log entry:**
  (p=FFMPEG, s=a.avi, timestamp=100, run=42, PRNG=17)
- Can simulate ***any*** schedule using log files
  - Including ***Offline Optimal*** (≈ dynamic prog. for BOUNDED KNAPSACK)

Dataset	Epoch	MAB algorithm	#bugs found for each belief				
			RPM	EWT	Density	Rate	RGR
Intra-Program	Fixed-Run	ϵ-Greedy	72	77	87	88	32
		Weighted-Random	72	84	84	**93**	85
		Uniform-Random			72		
		EXP3.S.1			58		
		Round-Robin			74		
	Fixed-Time	ϵ-Greedy	51	94	51	**109**	58
		Weighted-Random	67	94	58	100	108
		Uniform-Random			94		
		EXP3.S.1			95		
		Round-Robin			94		
Inter-Program	Fixed-Run	ϵ-Greedy	90	119	89	89	41
		Weighted-Random	90	131	92	**135**	94
		Uniform-Random			89		
		EXP3.S.1			72		
		Round-Robin			90		
	Fixed-Time	ϵ-Greedy	126	158	111	164	117
		Weighted-Random	152	157	100	**167**	165
		Uniform-Random			158		
		EXP3.S.1			161		
		Round-Robin			158		

Table 2: Comparison between scheduling algorithms.

Dataset	Epoch	MAB algorithm	#bugs found for each belief				
			RPM	EWT	Density	Rate	RGR
Intra-Program	Fixed-Run	ϵ-Greedy	72	77	87	88	32
		Weighted-Random	72	84	84	**93**	85
		Uniform-Random			72		
		EXP3.S.1			58		
		Round-Robin			74		
	Fixed-Tim	ϵ-Greedy	51	94	51	**109**	58
					58	100	108
					94		
					95		
					94		
Inter-Program	Fixed-Ru			9	89	89	41
				1	92	**135**	94
					89		
		EXP3.S.1			72		
		Round-Robin			90		
	Fixed-Time	ϵ-Greedy	126	158	111	164	117
		Weighted-Random	152	157	100	**167**	165
		Uniform-Random			158		
		EXP3.S.1			161		
		Round-Robin			158		

Table 2: Comparison between scheduling algorithms.

Recommendation 1: Use Weighted Random w/ Rate

Dataset	Epoch	MAB algorithm	#bugs found for each belief				
			RPM	EWT	Density	Rate	RGR
Intra-Program	Fixed-Run	ϵ-Greedy	72	77	87	88	32
		Weighted-Random	72	84	84	**93**	85
		Uniform-Random			72		
		EXP3.S.1			58		
		Round-Robin			74		
	Fixed-Time	ϵ-Greedy	51	94	51	**109**	58
					58	100	108
					94		
					95		
					94		
Inter-Program	Fixed-Run				89	89	41
					92	**135**	94
					89		
		EXP3.S.1			72		
		Round-Robin			90		
	Fixed-Time	ϵ-Greedy	126	158	111	164	117
		Weighted-Random	152	157	100	**167**	165
		Uniform-Random			158		
		EXP3.S.1			161		
		Round-Robin			158		

Table 2: Comparison between scheduling algorithms.

Recommendation 2: Use Fixed-Time Campaigns

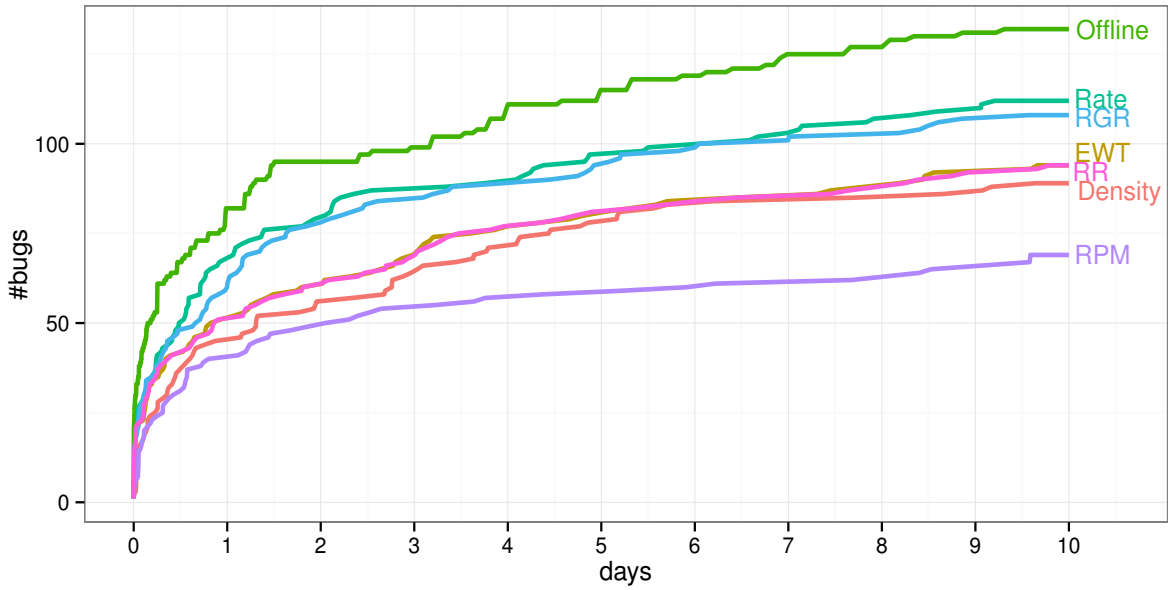
Comparison with CERT BFF v2.6

CERT BFF is the state-of-the-art fuzzing framework

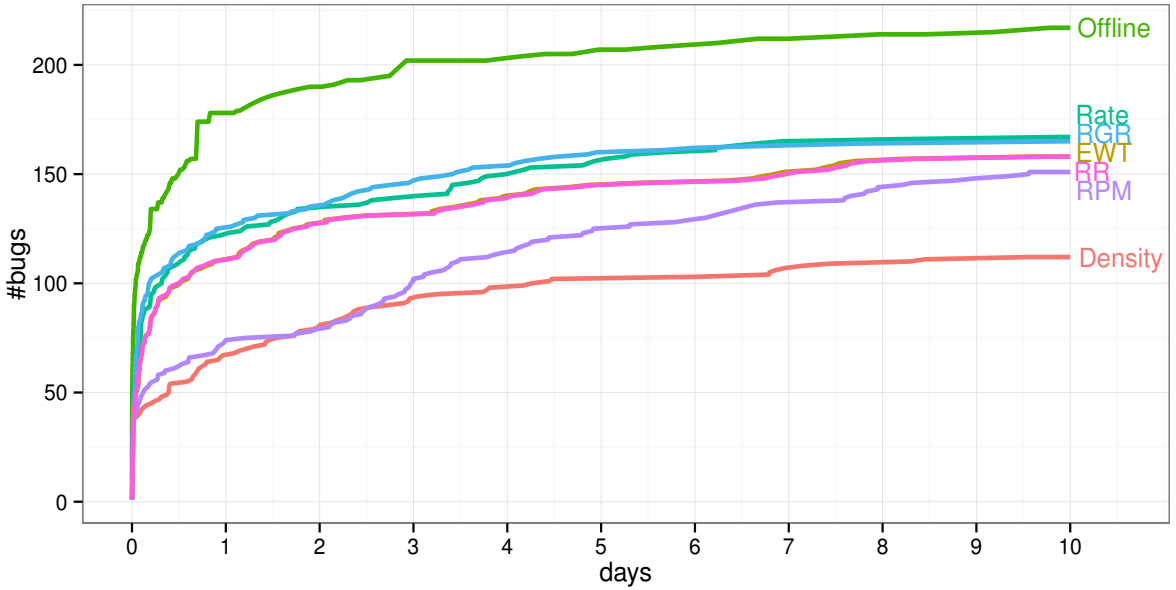
- Supports fuzzing ***one*** program w/ ***multiple*** seeds
- Varies mutation ratio ***online***
- ***Fixed-run*** epochs
- ***Weighted-Random*** MAB algorithm
- use ***Density*** ($^{\#\mathbf{bugs}}/_{\#\mathbf{runs}}$) as belief

Fixed-time Weighted-Random ***Rate*** finds on average 1.5x more bugs in our datasets (at a fixed mutation ratio)

Intra: FFMPEG Dataset



Inter: File Converters Dataset



Future Work

Vary mutation ratio

- m mutation ratios $\Rightarrow$ m-fold cost increase

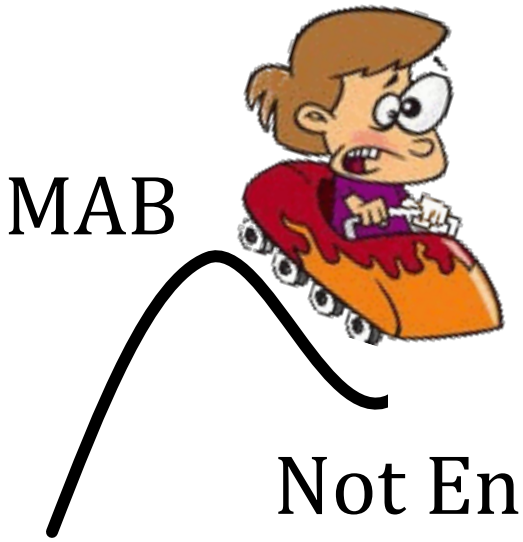
Online bug triage

- triage time is currently being discounted

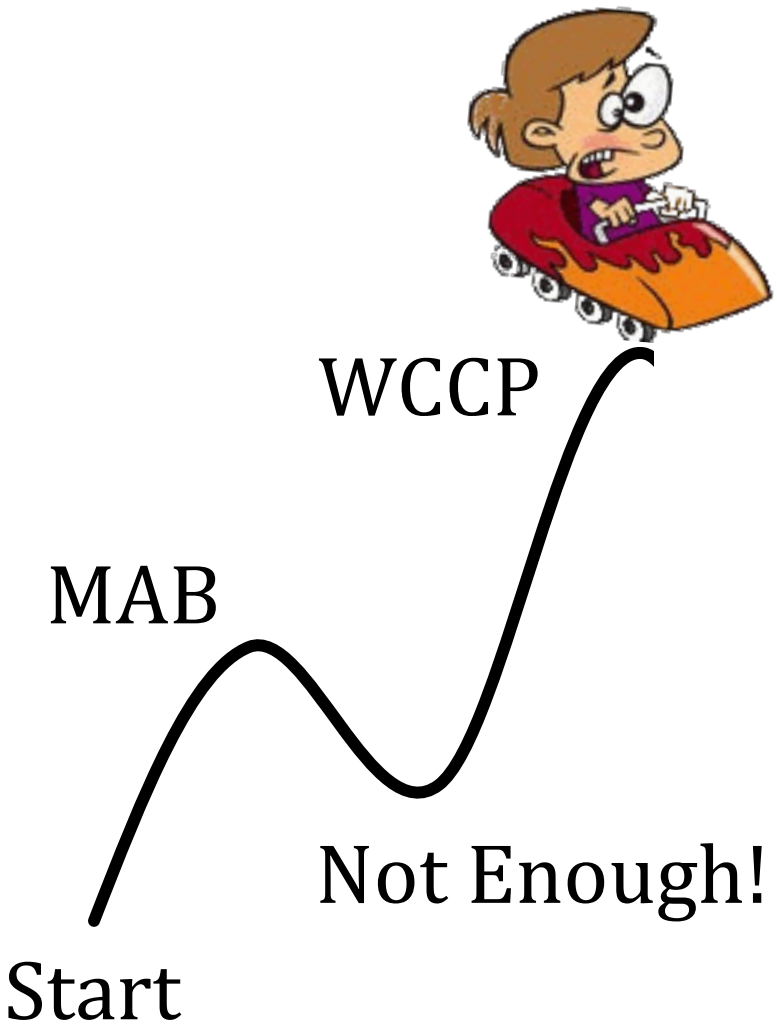
Other program testing techniques

- black-box generational (grammar-based) fuzzing?
- concolic execution?

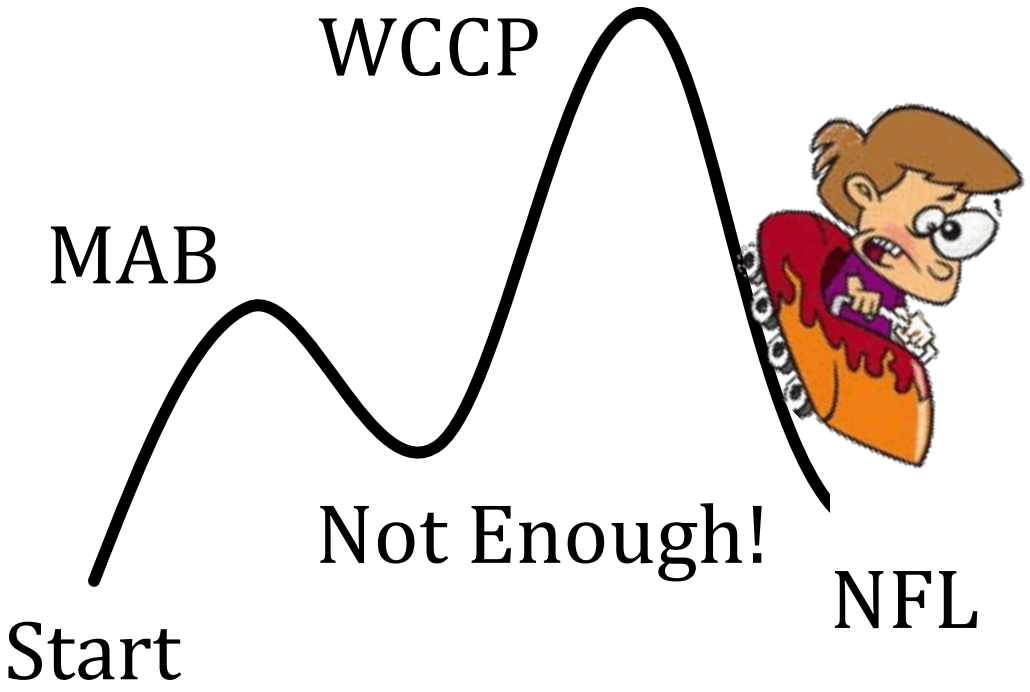
Summary



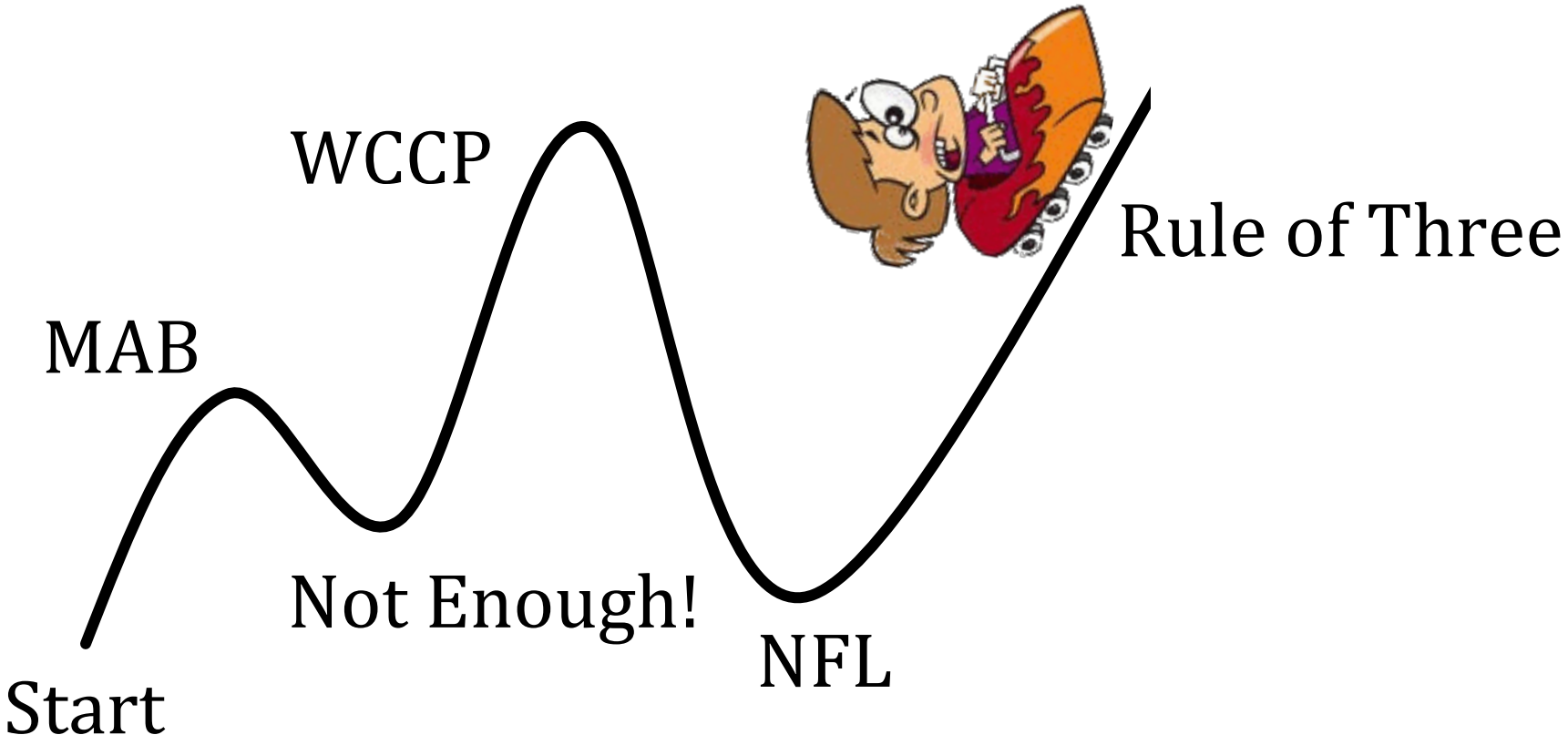
Summary



Summary



Summary



Summary

Algorithm Design

Rule of Three

WCCP

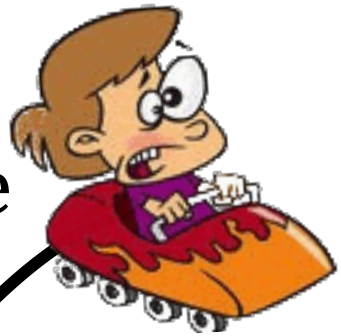
MAB

Not Enough!

NFL

Start

Summary



Open Science

Algorithm Design

Rule of Three

WCCP

MAB

Not Enough!

NFL

Start

http://security.ece.cmu.edu/fuzzsim/

FuzzSim[1]: Black-box Fuzzing Simulator

Black-box mutational fuzzing is an effective, albeit simple, way to find bugs in software. FuzzSim is a black-box fuzzing scheduling simulator that can test various seed selection algorithms. The main purpose of this tool is to ask the following question: given a set of programs and seeds, what is the best way to schedule fuzzing of the programs on the seeds to maximize the number of unique bugs found within a fixed amount of time?

Installation

Download FuzzSim

After downloading

```
$ tar xvfz fuzzs
$ cd fuzzsim-0.1
$ ./configure
$ make
```

Quick Start

To see the usage, type:

FuzzSim Download Page

Download files from the below:

fuzzsim.0.1.tgz (MD5: 0420abf52d6d74014fac8607fedc99ea)
fuzzdata-intra.tgz (MD5: 3dd151054c01e14d39e25eda0e48dd35)
fuzzdata-inter.tgz (MD5: 0a3ec7c2b1550812a87d517cbf2e82c2)