USENIX Security 2023

autofz:

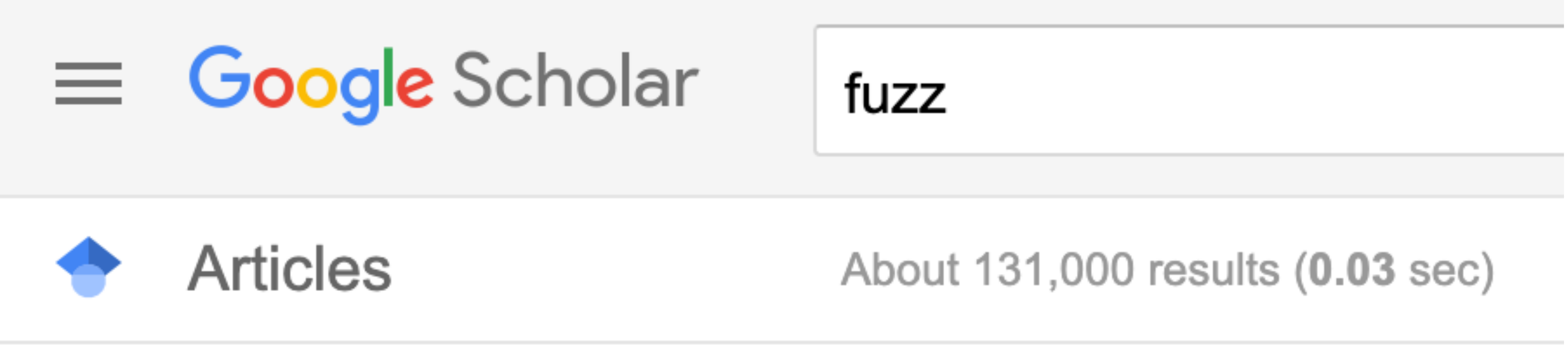
Automated Fuzzer Composition at Runtime

Yu-Fu Fu, Jaehyuk Lee, Taesoo Kim

SSLab @GeorgiaTech

Fuzzing Wars: A Flood of Different Fuzzers

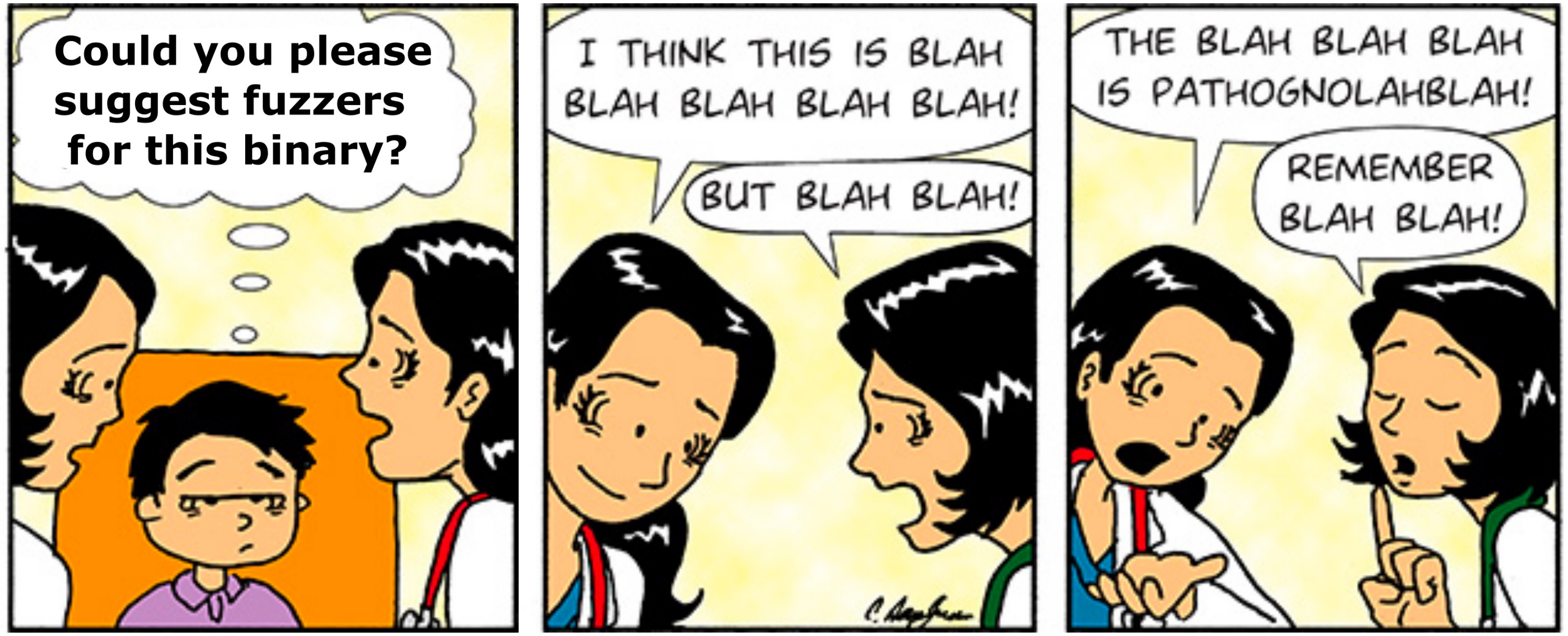
- Fuzzing is all about efficiently producing input that can uniquely locate bugs
- Various fuzzing techniques ⇒ **tons of different fuzzers** in the wild
  - Symbolic execution, Taint analysis, or even Machine Learning for fuzzing



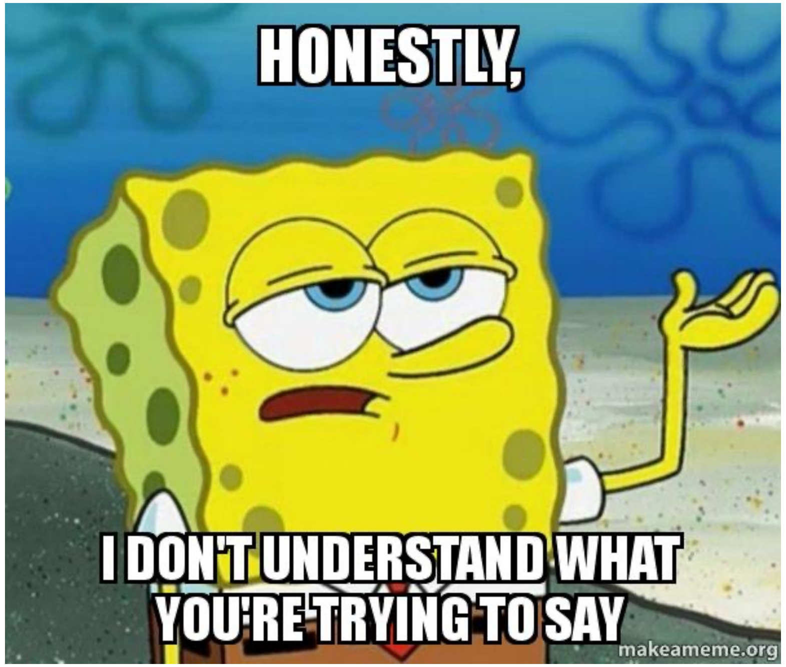
fuzz

About 131,000 results (**0.03** sec)

Lost in the Fuzz: Selection Burden in Modern Fuzzing

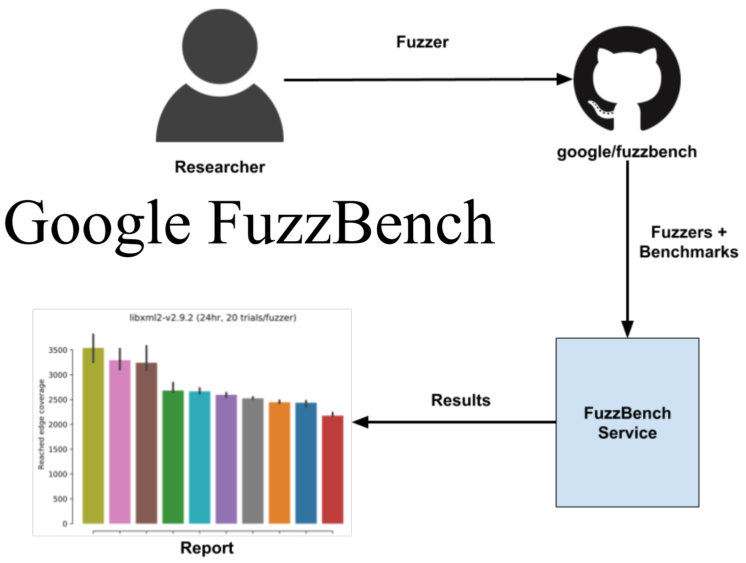

Could you please suggest fuzzers for this binary?
I THINK THIS IS BLAH BLAH BLAH BLAH BLAH!
BUT BLAH BLAH!
THE BLAH BLAH BLAH IS PATHOGNOLAHBLAH!
REMEMBER BLAH BLAH!

Original Image: Carlo Jose San Juan, MD



- Okay, **as a user**, which fuzzer should I use to get the best result?
- Most users don’t have knowledge about details of each fuzzer

Community Solution: Fuzzing Benchmark!



Google FuzzBench

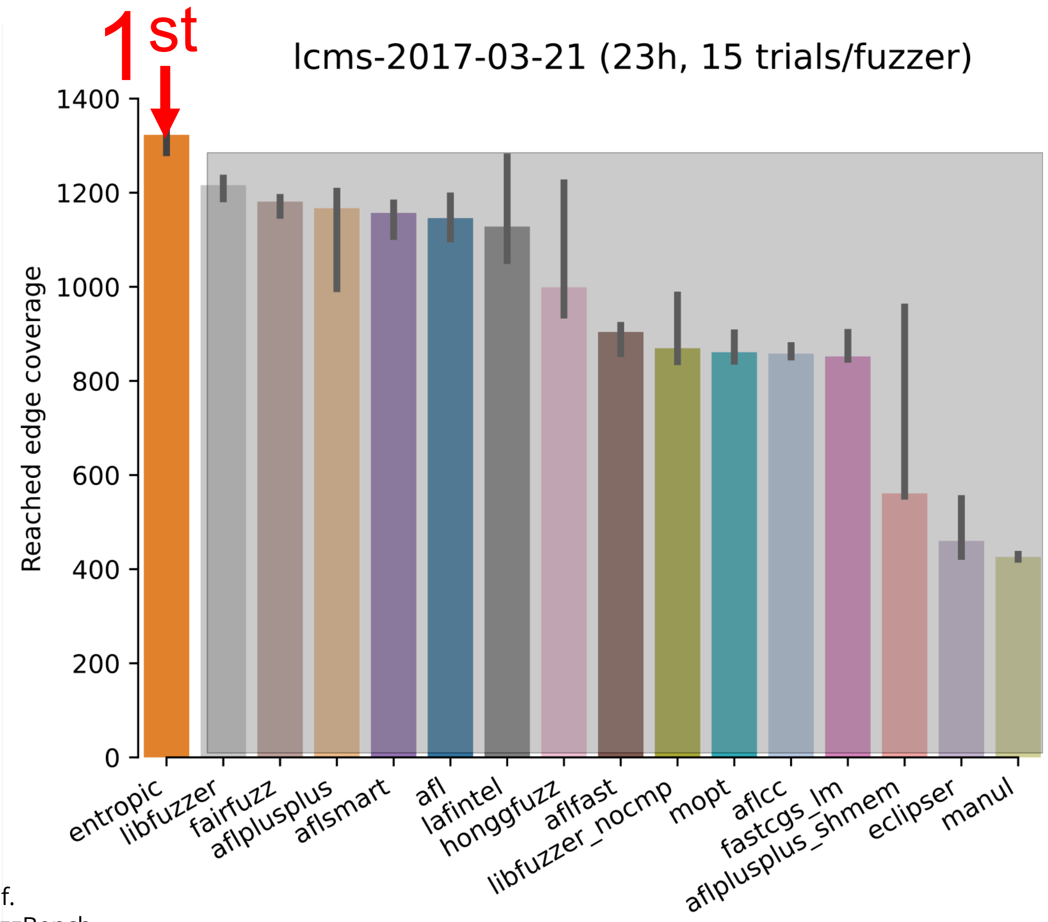
Magma

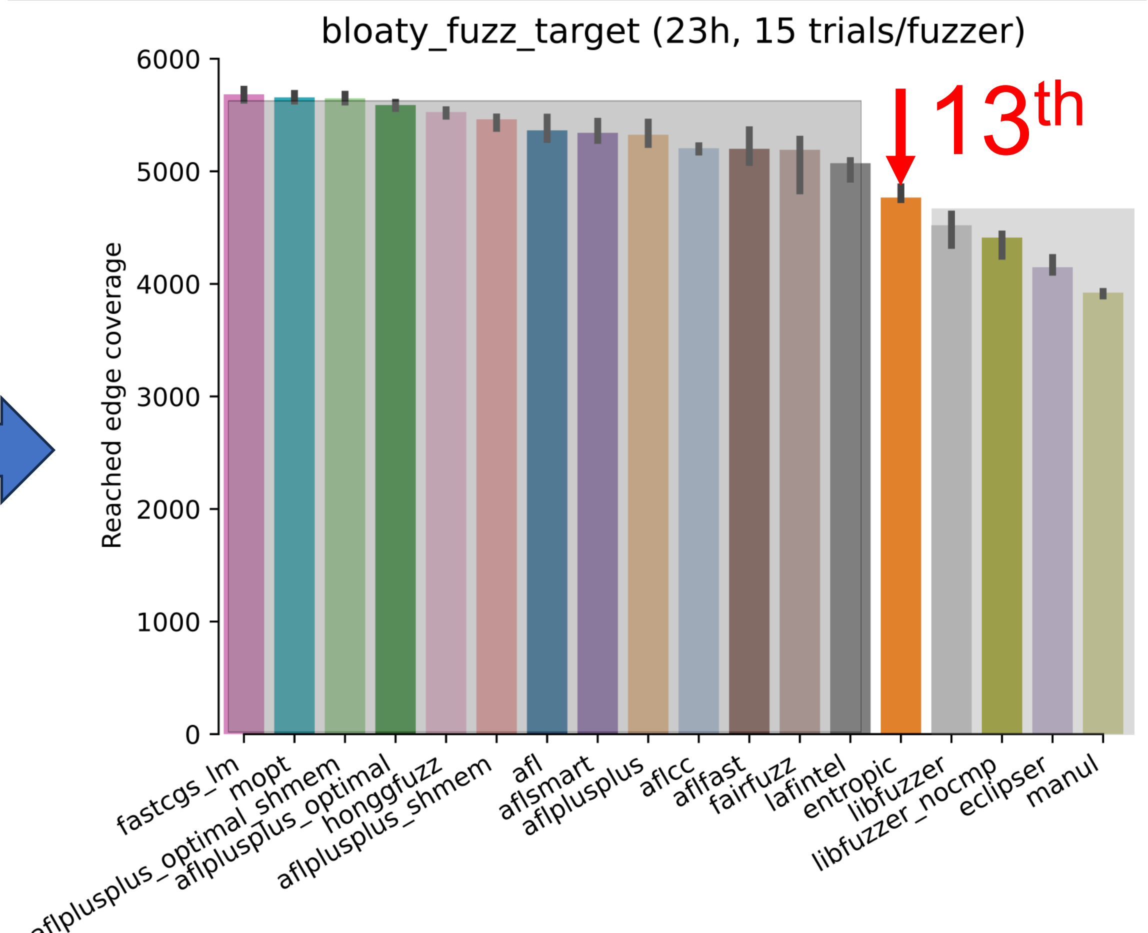
- Fuzzing benchmark: creating a set of standard benchmarks for fuzzing!
  - Compare the performance of fuzzers for a wide range of applications
  - Choose the one performing **best on average** across the benchmarks

- The result is **not always an optimal decision** for every target!
- It does not guarantee the best outcome for the targets not in the benchmark (overfitting)

Biases in Selection: Target-Dependent Performance

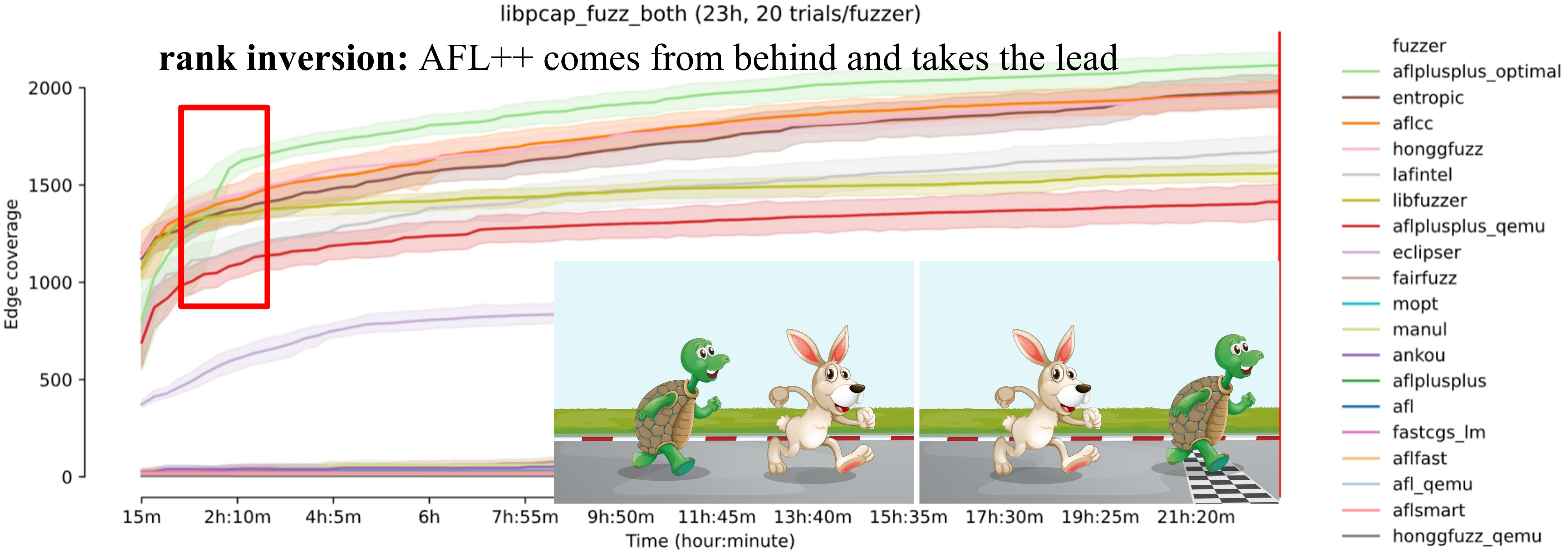
- No universal fuzzer invariably outperforms others
- The performance of fuzzers can significantly **differ depending on the target**





Ref.
FuzzBench

Biases in Selection: Inconsistent Performance at Runtime



rank inversion: AFL++ comes from behind and takes the lead

- The efficiency of each fuzzer **fluctuates** throughout its execution
- **No guarantee** that initially well-performing fuzzer will be **the final winner**
- Rank is **consistent in short time**

Ref. FuzzBench

Rule of Thumb: Past Success is No Guarantee of Future Results

- Benchmark results cannot ensure that selected fuzzer will be effective in fuzzing user’s binary

- Using a static fuzzer selection can result in **suboptimal** outcomes
  - performance bias & rank inversion during runtime

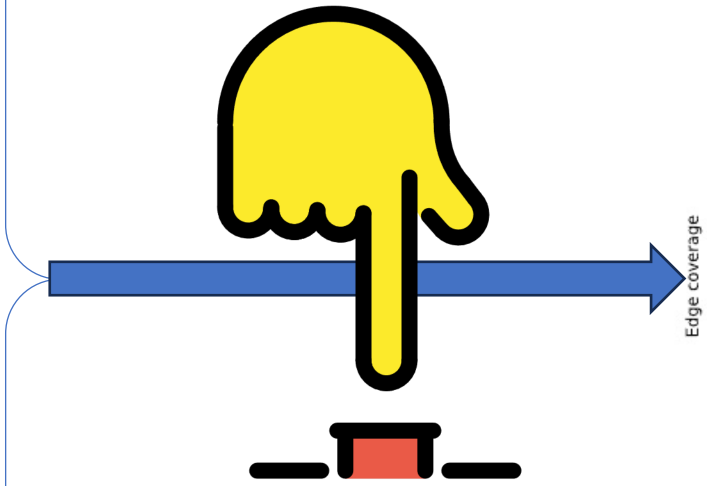
- Relying **solely on static information** is the cause!

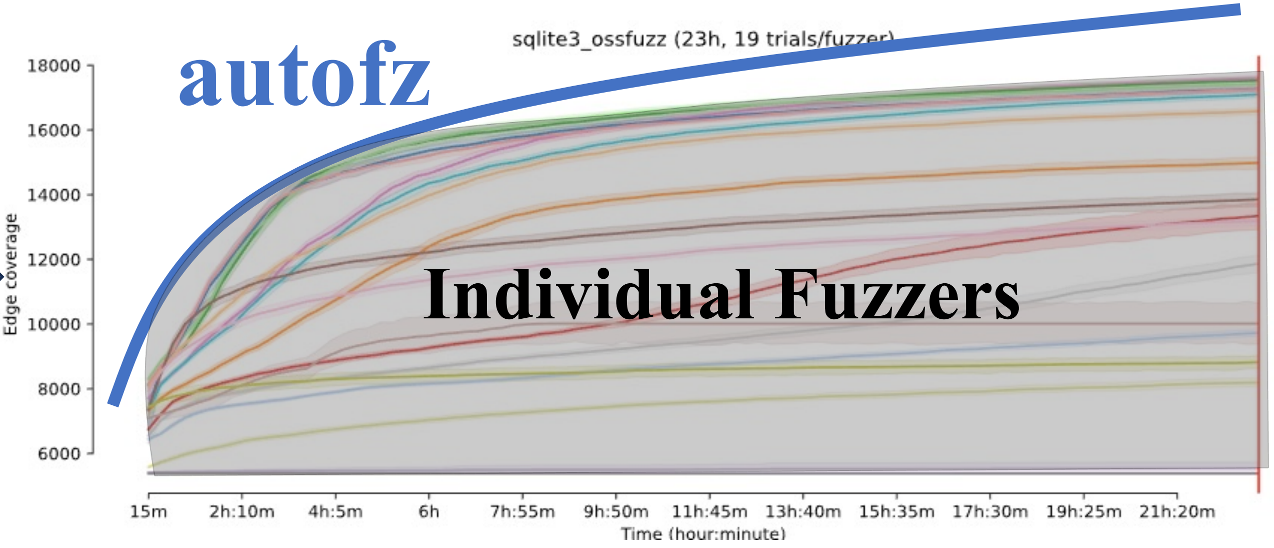
Dynamic Composition of Fuzzers as a Push-button Solution

List of fuzzers

- aflplusplus
- fastcgs_lm
- aflsmart
- afl
- mopt
- aflplusplus_optimal
- aflfast
- aflplusplus_qemu
- honggfuzz
- lafintel
- honggfuzz_qemu
- fairfuzz
- afl_qemu
- entropic
- libfuzzer
- manul
- eclipser

No fuzzing expertise or **benchmarking** is necessary.
Provide list of fuzzers and push the button! That's all!





autofz

Individual Fuzzers

autofz **automatically** deploys a set of fuzzer(s)
Outperforms the best individual fuzzers **in any target**

Utilizing Runtime Information (Trend) in Selection

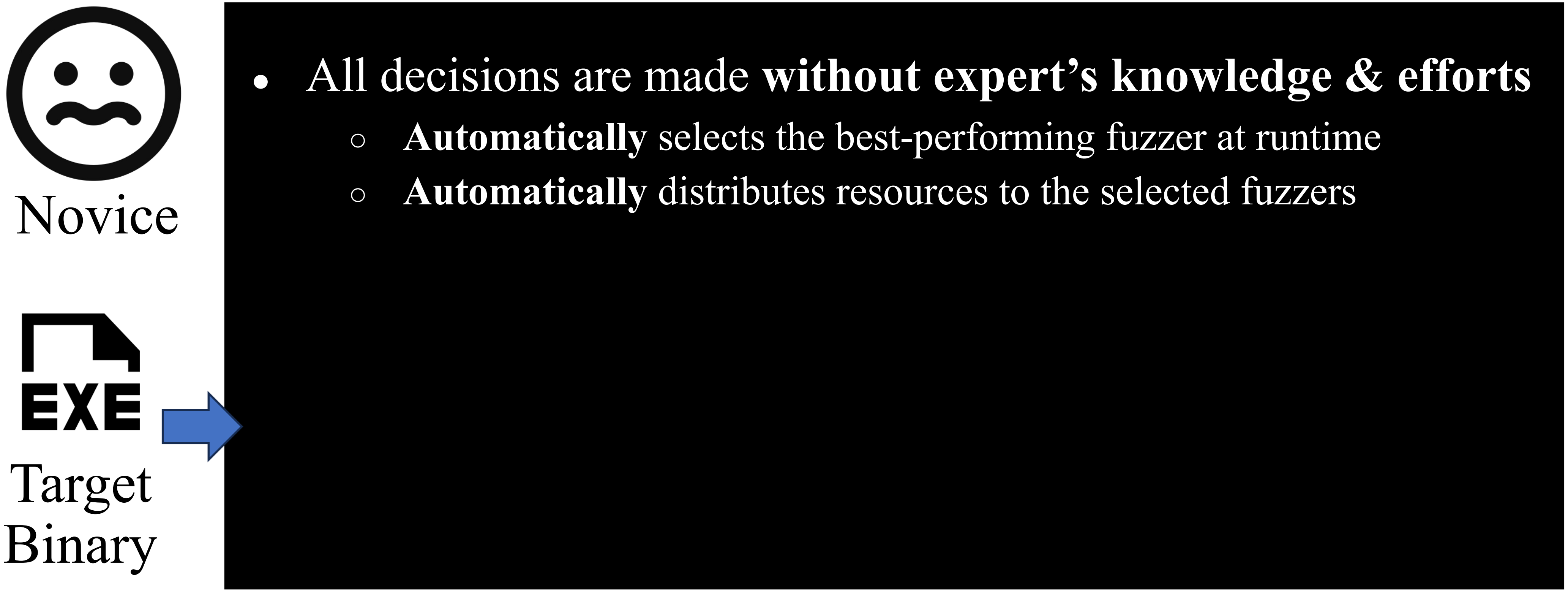
autofz as a **BLACK BOX** to user

Novice

Target Binary

Utilizing Runtime Information (Trend) in Selection

autofz as a **BLACK BOX** to user



Novice

Target Binary

- All decisions are made **without expert's knowledge & efforts**
    - **Automatically** selects the best-performing fuzzer at runtime
    - **Automatically** distributes resources to the selected fuzzers

Utilizing Runtime Information (Trend) in Selection

autofz as a **BLACK BOX** to user

Novice

Target Binary

- All decisions are made **without expert's knowledge & efforts**
  - **Automatically** selects the best performing fuzzer at runtime
  - **Automatically** distributes resources to the selected fuzzers
- How? autofz utilizes **runtime trend** of fuzzers!
  - **Runtime Trend:** runtime progress of fuzzers in short time
  - Select well-performing fuzzer(s) based on the **runtime trends**
  - Distribute resources to selected fuzzer(s) based on the **runtime trends**

Utilizing Runtime Information (Trend) in Selection

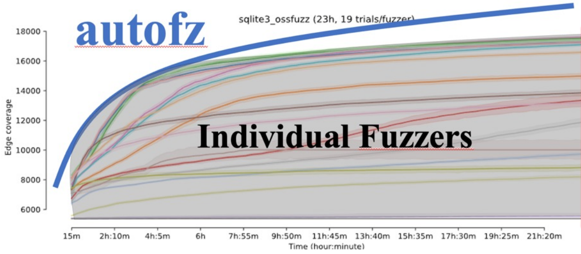
autofz as a **BLACK BOX** to user

Novice

Target Binary

- All decisions are made **without expert's knowledge & efforts**
  - **Automatically** selects the best performing fuzzer at runtime
  - **Automatically** distributes resources to the selected fuzzers
- How? autofz utilizes **runtime trend** of fuzzers!
  - **Runtime Trend:** runtime progress of fuzzers in short time
  - Select well-performing fuzzer(s) based on the **runtime trends**
  - Distribute resources to selected fuzzer(s) based on the **runtime trends**

Expert-level Outcome

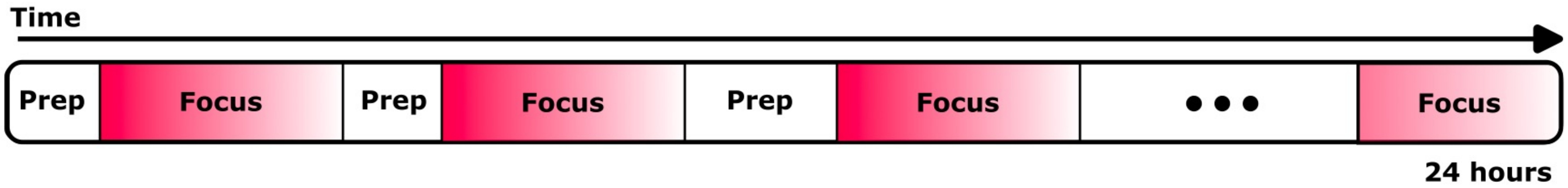


How to Effectively Capture/Utilize Runtime Trends?

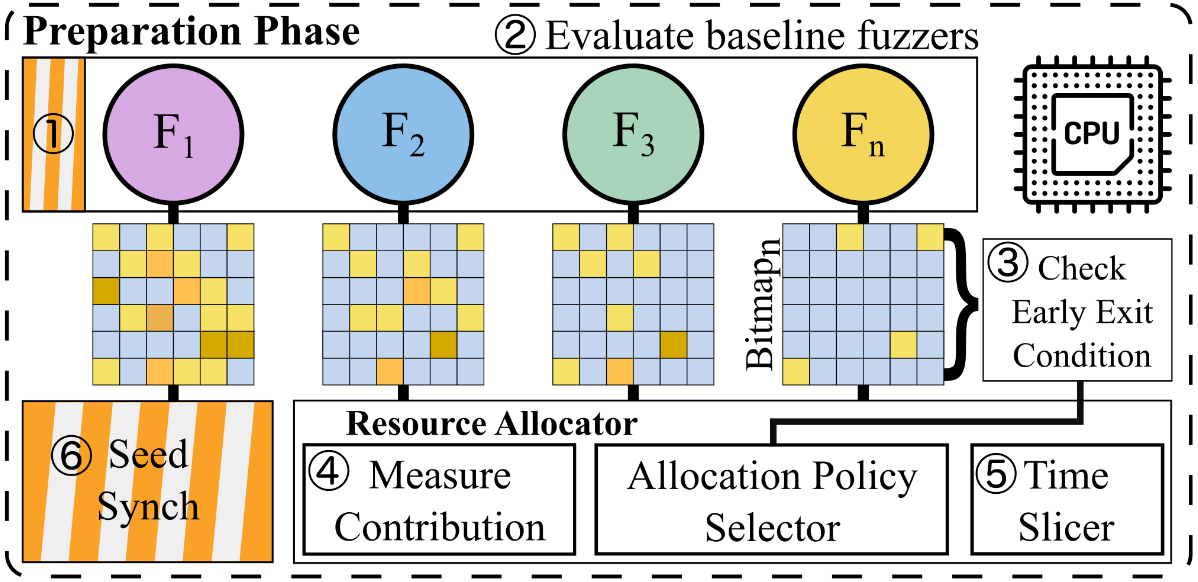
- We use **trend as feedback** in fuzzer selection and utilization!
  - Fuzzer showing strong trend is more likely to be good at finding more bugs

- As fuzzing progresses, the runtime trend can be **changed**
  - Repeatedly measure the runtime trend in short time period

- **Two-phase algorithm**: split entire fuzzing run into multiple rounds of measurement (preparation) and execution (focus)

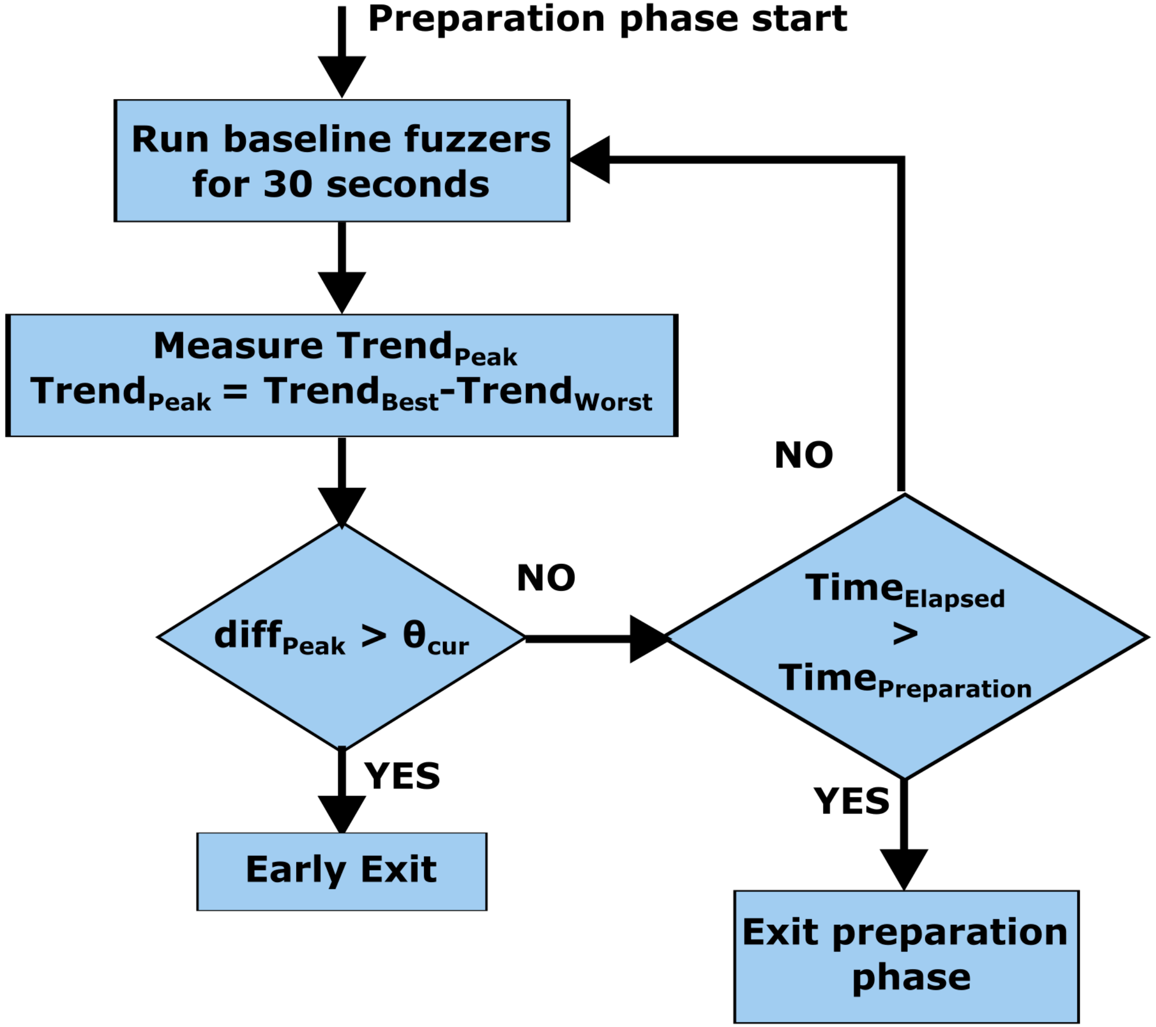


Preparation Phase



- Run each fuzzer for **small time frame** (minimal overhead in measuring trends)
- Trend is measured by **unique coverage** discovered in the time window
  - AFL Bitmap to measure the unique coverage
- Select fuzzers and distribute resources (CPU) based on the trends
- Early Exit: optimization for reducing resource waste in preparation phase
  - Terminate preparation phase **as soon as we find outstanding fuzzer(s)**

Preparation Phase: Outstanding Fuzzer & Early Exit

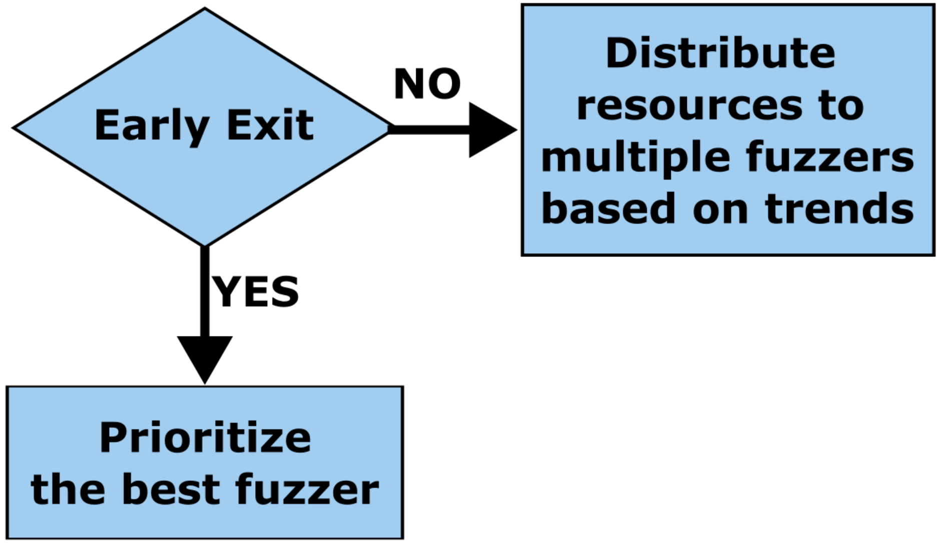

Preparation phase start
Run baseline fuzzers for 30 seconds
Measure $\text{Trend}_{\text{Peak}}$
$\text{Trend}_{\text{Peak}} = \text{Trend}_{\text{Best}} - \text{Trend}_{\text{Worst}}$
$\text{diff}_{\text{Peak}} > \theta_{\text{cur}}$
NO
YES
Early Exit
$\text{Time}_{\text{Elapsed}} > \text{Time}_{\text{Preparation}}$
NO
YES
Exit preparation phase

- Preparation should run all fuzzers to measure trends

- Preparation phase early-exits when there is outstanding fuzzer
    - Minimize overhead incurred by running all fuzzers

- Measures peak difference of trends and compares it with predefined threshold
    - If peak difference > threshold, early exit
    - Threshold is automatically configured at runtime

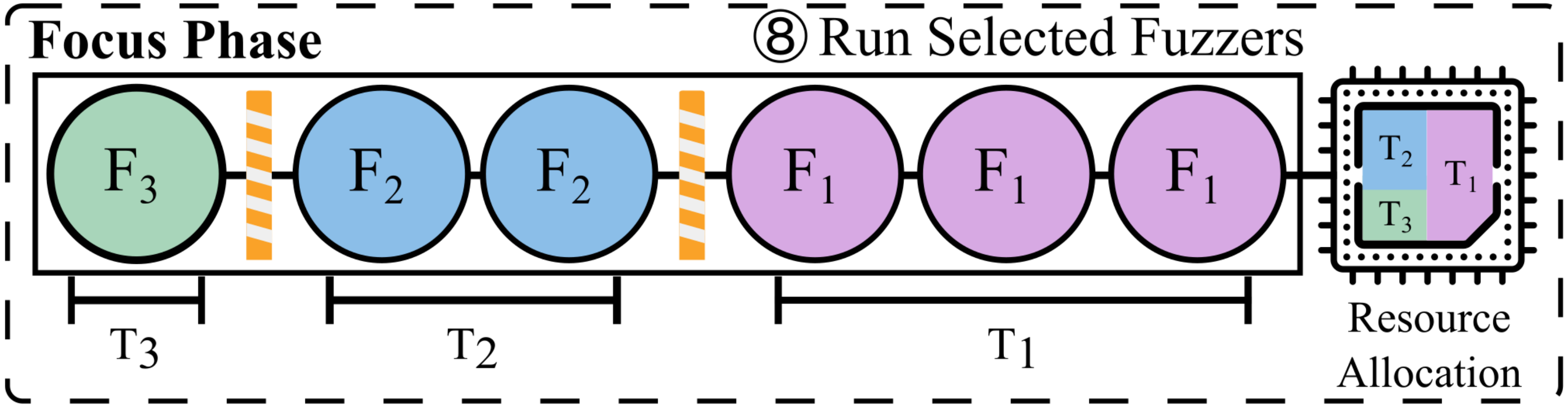
Preparation Phase: Resource Assignment Algorithm


Early Exit
NO
Distribute resources to multiple fuzzers based on trends
YES
Prioritize the best fuzzer

- Two resource allocation strategies
  - Individual fuzzer outperforms others ⇒ Assign entire resources to outperforming one
  - No outstanding fuzzer ⇒ Distribute resources to multiple fuzzers based on trends

- Best strategy will be selected based on early exit (automatically)

Focus Phase

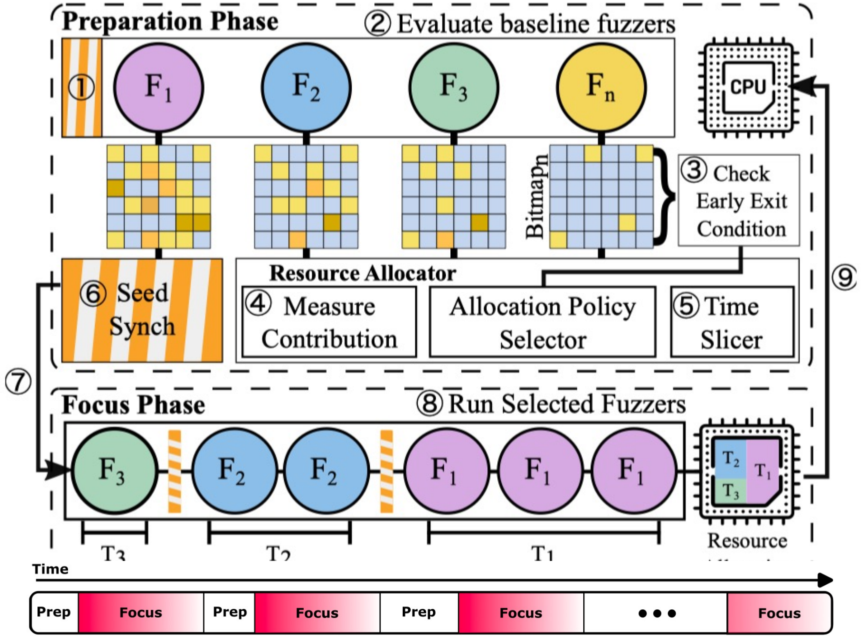


- Run selected fuzzers based on allocation metadata

- Number of fuzzers executed during the focus phase can vary
  - Sole individual (best) fuzzer
  - Combination of multiple different fuzzers

- CPU time allocated for each fuzzer can be different
  - It can prioritize specific fuzzers based on the contribution of each fuzzer

Why autofz can do better than others?



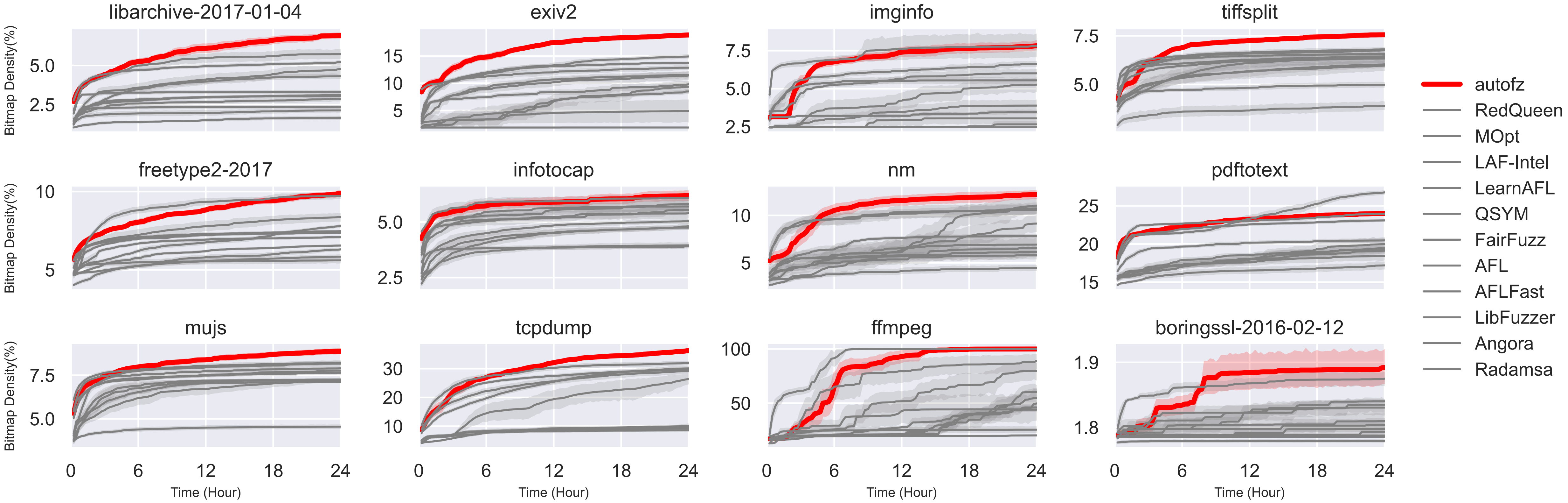
- **Two-phase design** captures trend accurately
  - autofz can tell which fuzzer(s) perform well during specific time periods
  - Can achieve optimal result by deploying the best performing fuzzer **at the right time**

- **Resource Distribution:** Survival of the fittest!
  - autofz **gives priority to effective fuzzers** while giving lower priority to less effective
  - Takes benefit of **individual fuzzer** and **combination of different fuzzers**

Evaluation Setting

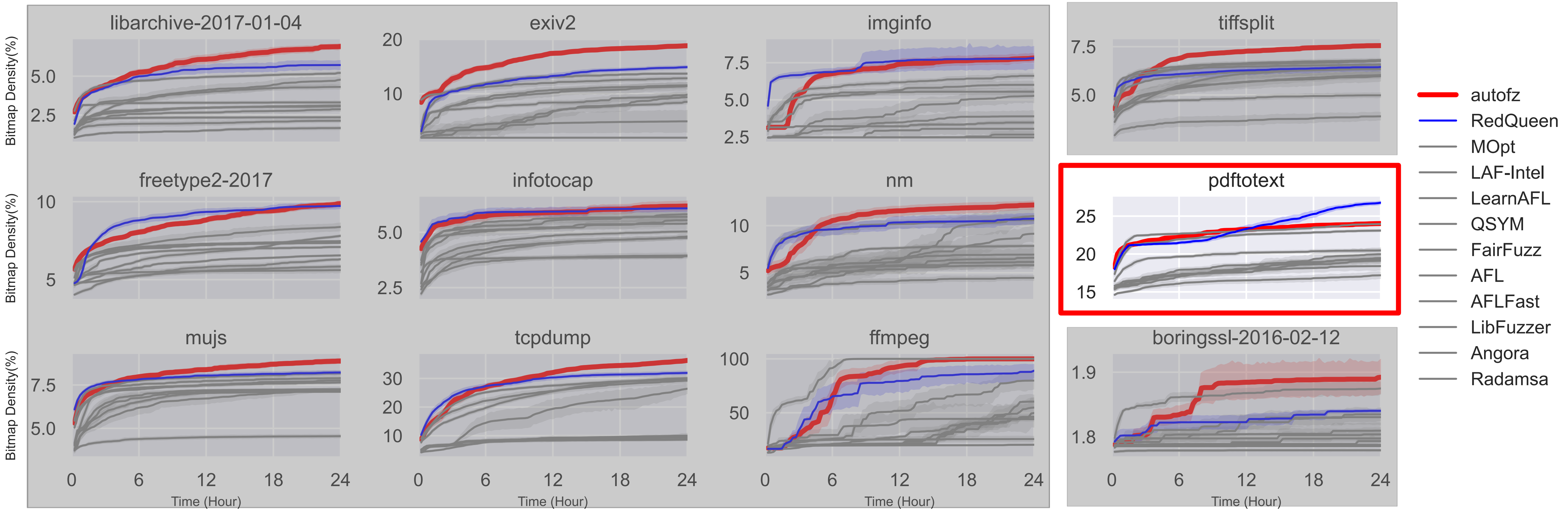
- 11 fuzzers
  - AFL, MOpt, FairFuzz, AFLFast, LearnAFL
  - RedQueen, LAF-Intel, QSYM, Angora
  - Radamsa
  - LibFuzzer (only for FTS)
- 2 benchmark
  - UNIFUZZ
  - Fuzzer Test Suite (FTS)
- 24 hours
- 10 repetitions

autofz vs. other fuzzers (coverage)



Top in 11/12 programs

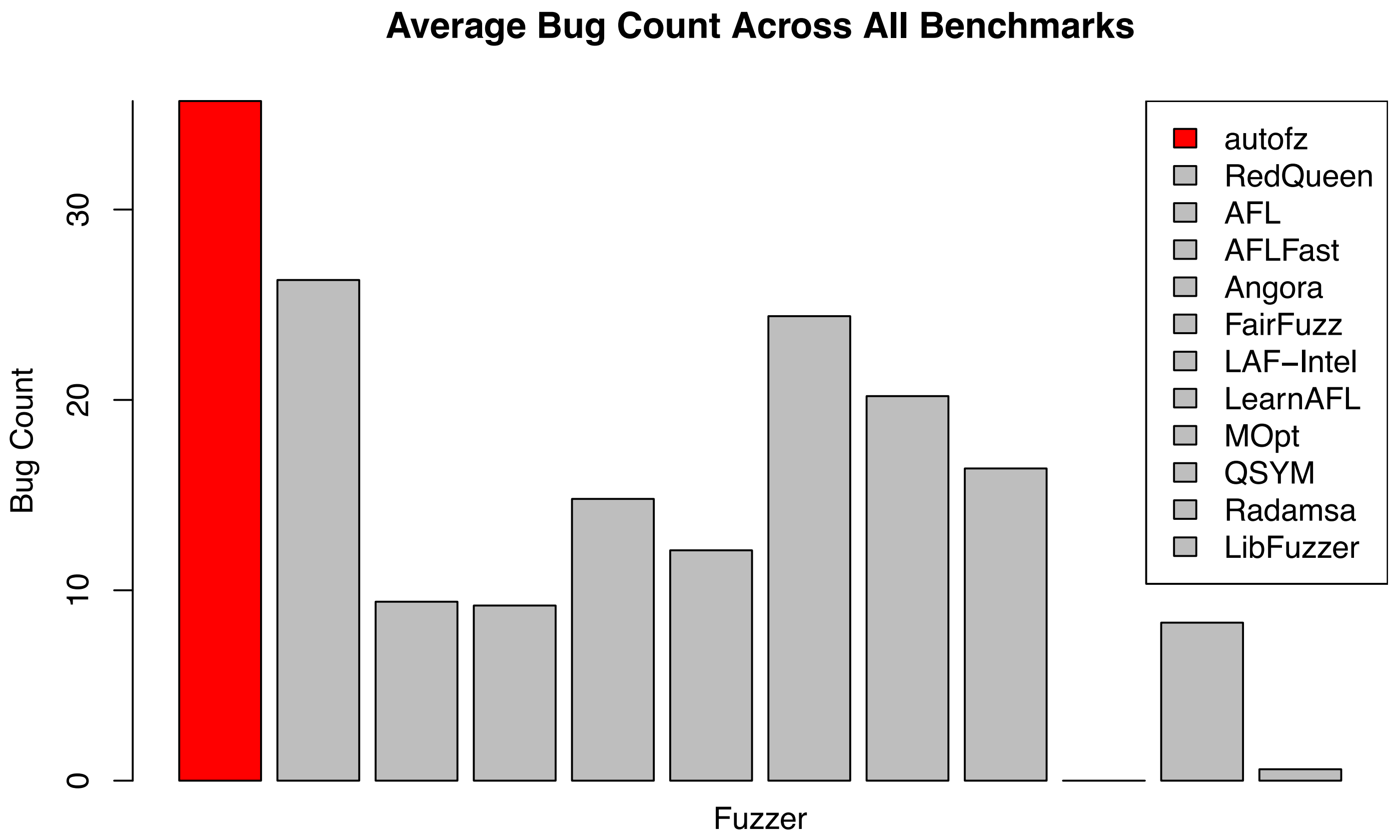
autofz vs. other fuzzers (coverage) – pdftotext case



RedQueen needs to accumulate more **internal states (> 12 hours)** to have better performance, but this **does not reflect on** its coverage, so autofz does not prioritize it by design.

It is a super **rare** case during our evaluation.

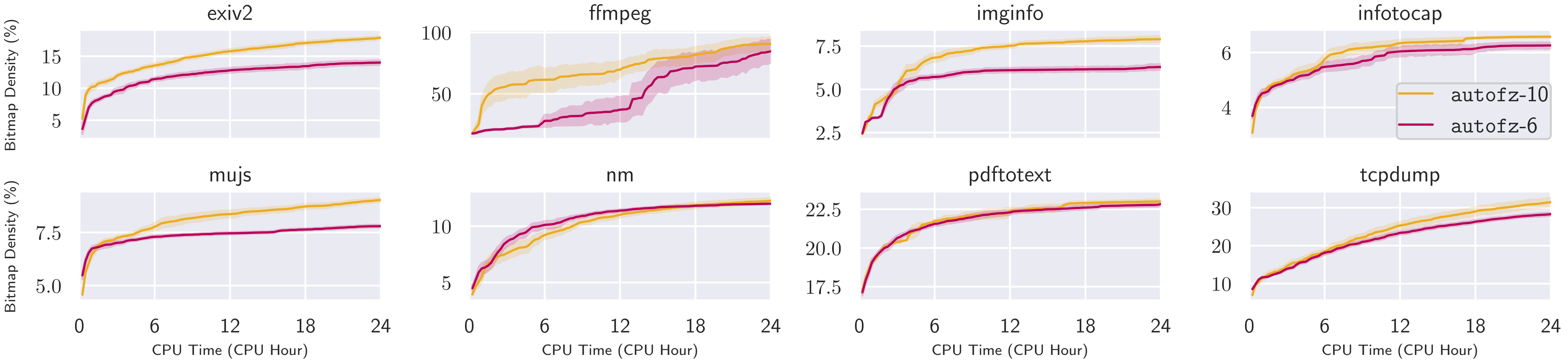
autofz vs. individual fuzzers (bugs)



Number of bugs = (Total number of bugs found in 10 rounds) / 10

autofz finds most bugs

Bring More Fuzzers → Better Result



autofz-6 = [AFL, FairFuzz, QSYM, AFLFast, LAF-INTEL, Radamsa] **autofz-10** = [All baseline fuzzers except libFuzzer]

- Gains: **Diversity** of fuzzers can facilitate the exploration of challenging-to-reach paths
- Losses: run more (possibly bad) fuzzers to measure their trends (in preparation phase)
  - minimized by resource allocation algorithm in focus phases

Gains > Losses when adding fuzzers

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Conclusion

- **Non-expert users** can fully take advantage of fuzzing to make their software more secure
- autofz can **bridge the gap** between developing new fuzzers and their effective deployment (**without** running **benchmarks** first)
- Just bring **more** fuzzers! We will give you **better** results!

SSLab @GeorgiaTech

paper

code