Seed Selection for Successful Fuzzing

Adrian Herrera, Hendra Gunadi, Shane Magrath,
Michael Norrish, Mathias Payer, Antony L. Hosking

whoami

- PhD student at the Australian National University
- Interests in fuzzing, binary analysis, program analysis

Australian National University

DST GROUP

DATA 61

Seed Selection for Successful Fuzzing

Adrian Herrera
ANU & DST
Australia

Hendra Gunadi
ANU
Australia

Shane Magrath
DST
Australia

Michael Norrish
CSIRO's Data61 & ANU
Australia

Mathias Payer
EPFL
Switzerland

Antony L. Hosking
ANU & CSIRO's Data61
Australia

ABSTRACT

Mutation-based greybox fuzzing—unquestionably the most widely-used fuzzing technique—relies on a set of non-crashing seed inputs (a corpus) to bootstrap the bug-finding process. When evaluating a fuzzer, common approaches for constructing this corpus include: (i) using an empty file; (ii) using a single seed representative of the target's input format; or (iii) collecting a large number of seeds (e.g., by crawling the Internet). Little thought is given to *how* this seed choice affects the fuzzing process, and there is no consensus on which approach is best (or even if a best approach exists).

To address this gap in knowledge, we systematically investigate and evaluate how seed selection affects a fuzzer's ability to *find bugs in real-world software*. This includes a systematic review of seed selection practices used in both evaluation and deployment contexts, and a large-scale empirical evaluation (over 33 CPU-years) of six seed selection approaches. These six seed selection approaches include three *corpus minimization* techniques (which select the smallest subset of seeds that trigger the same range of instrumentation data points as a full corpus).

Our results demonstrate that fuzzing outcomes vary significantly depending on the initial seeds used to bootstrap the fuzzer, with minimized corpora outperforming singleton, empty, and large (in the order of thousands of files) seed sets. Consequently, we encourage seed selection to be foremost in mind when evaluating/deploying fuzzers, and recommend that (a) seed choice be carefully considered and explicitly documented, and (b) never to evaluate fuzzers with only a single seed.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software and application security**.

KEYWORDS

fuzzing, corpus minimization, software testing

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07.
https://doi.org/10.1145/3460319.3464795

ACM Reference Format:
Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464795

1 INTRODUCTION

Fuzzing is a dynamic analysis technique for finding bugs and vulnerabilities in software, triggering crashes in a target program by subjecting it to a large number of (possibly malformed) inputs. *Mutation-based* fuzzing typically uses an initial set of valid seed inputs from which to generate new seeds by random mutation. Due to their simplicity and ease-of-use, mutation-based greybox fuzzers such as AFL [74], honggfuzz [64], and libFuzzer [61] are widely deployed, and have been highly successful in uncovering thousands of bugs across a large number of popular programs [6, 16]. This success has prompted much research into improving various aspects of the fuzzing process, including mutation strategies [39, 42], energy assignment policies [15, 25], and path exploration algorithms [14, 73]. However, while researchers often note the importance of high-quality input seeds and their impact on fuzzer performance [37, 56, 58, 67], few studies address the problem of *optimal design and construction of corpora* for mutation-based fuzzers [56, 58], and none assess the precise impact of these corpora in coverage-guided mutation-based greybox fuzzing.

Intuitively, the collection of seeds that form the initial corpus should generate a broad range of observable behaviors in the target. Similarly, candidate seeds that are behaviorally similar to one another should be represented in the corpus by a single seed. Finally, both the total size of the corpus and the size of individual seeds should be minimized. This is because previous work has demonstrated the impact that file system contention has on industrial-scale fuzzing. In particular, Xu et al. [71] showed that the overhead from opening/closing test-cases and synchronization between workers each introduced a 2× overhead. Time spent opening/closing test-cases and synchronization is time diverted from mutating inputs and expanding code coverage. Minimizing the total corpus size and the size of individual test-cases reduces this wastage and enables time to be (better) spent on finding bugs.

Under these assumptions, simply gathering as many input files as possible is not a reasonable approach for constructing a fuzzing corpus. Conversely, these assumptions also suggest that beginning with the "empty corpus" (e.g., consisting of one zero-length file) may be less than ideal. And yet, as we survey here, the majority of published research uses either (a) the "singleton corpus" (e.g., a single seed representative of the target program's input format),

What is Fuzzing?

Automated program testing technique

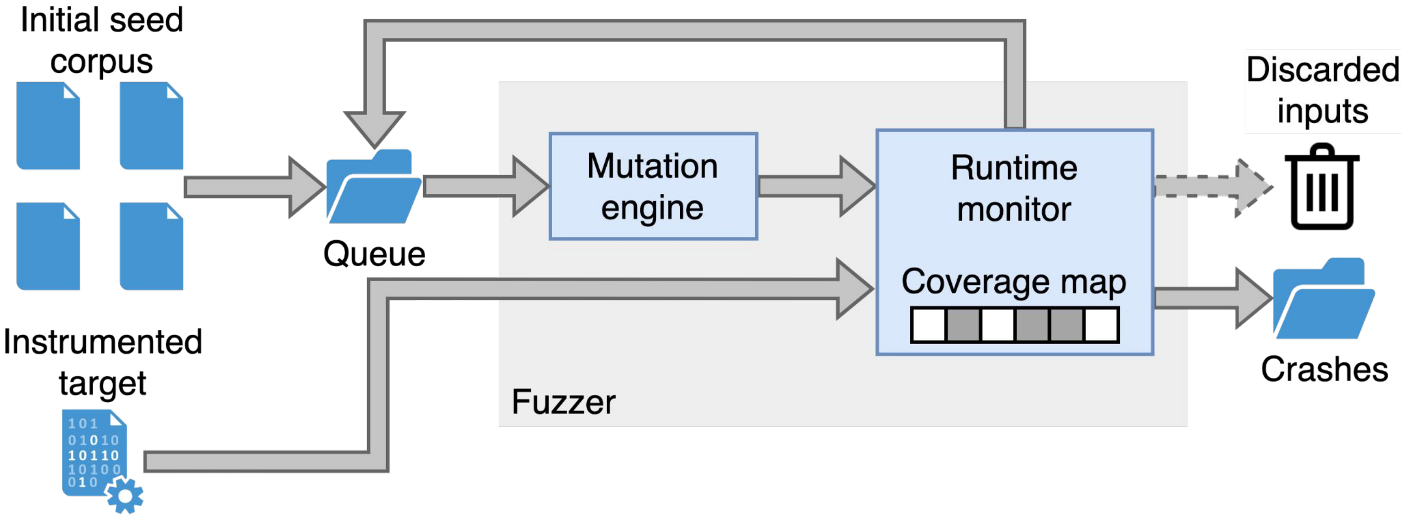
1. Feed your program malformed inputs
2. Monitor your program for crashes
3. Return to 1.

Is that it?

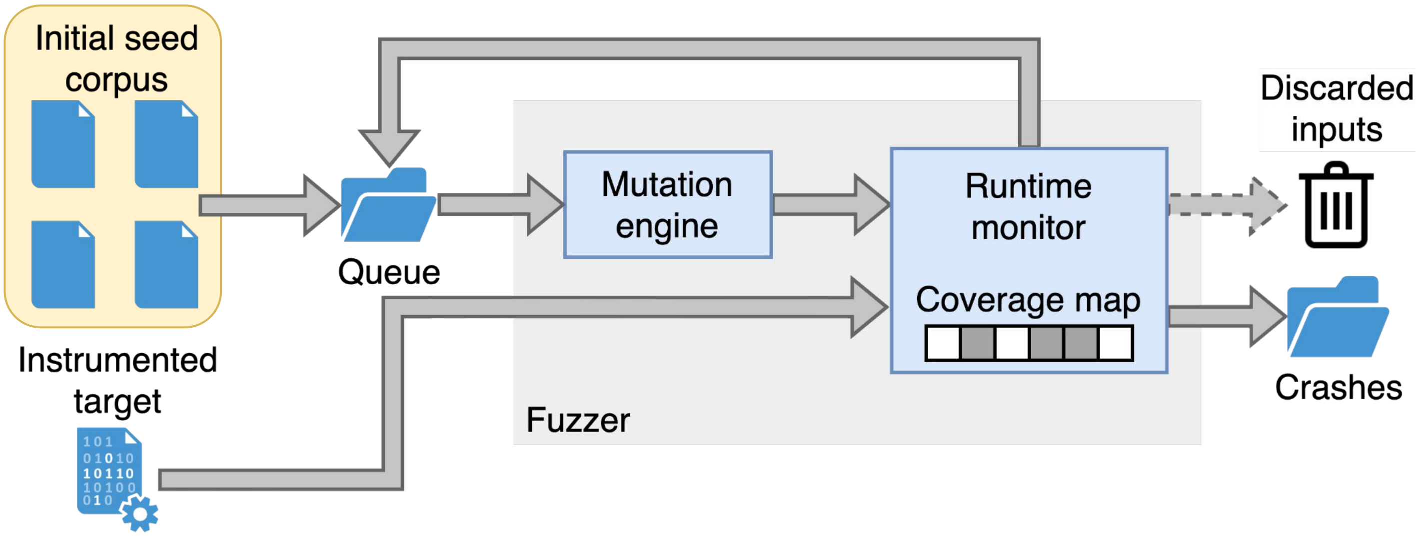
Is that it?

Not quite!

A Generic Mutational Greybox Fuzzer



A Generic Mutational Greybox Fuzzer



How to select these seeds? **Why** does it matter?

Seed Selection Practices

From “Evaluating Fuzz Testing”, Klees et al.

> *“Most papers treated the choice of seeds casually, apparently assuming that any seed would work equally well, without providing particulars.”*

Australian National University

Seed Selection Practices

Since 2018

- 3 studies **do not report seeds**
- 7 studies use **benchmark/fuzzer-provided seeds**
- 2 studies use **manually-constructed seeds**
- 5 studies use **random seeds**
  - 2 studies use a **corpus minimization** tool
- 8 studies use the **empty seed**

Does seed choice matter?

Australian National University

A Reproduction Experiment: *RedQueen*

Initial corpus

> *"Unless stated otherwise, we used an uninformed, generic seed consisting of different characters from the printable ASCII set"*

`ABC...XYZabc...xyz012...789!"$...~+*`

REDQUEEN: Fuzzing with Input-to-State Correspondence

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik and Thorsten Holz
Ruhr-Universität Bochum

***Abstract*—Automated software testing based on fuzzing has experienced a revival in recent years. Especially feedback-driven fuzzing has become well-known for its ability to efficiently perform randomized testing with limited input corpora. Despite a lot of progress, two common problems are magic numbers and (nested) checksums. Computationally expensive methods such as taint tracking and symbolic execution are typically used to overcome such roadblocks. Unfortunately, such methods often require access to source code, a rather precise description of the environment (e.g., behavior of library calls or the underlying OS), or the exact semantics of the platform's instruction set.**

In this paper, we introduce a lightweight, yet very effective alternative to taint tracking and symbolic execution to facilitate and optimize state-of-the-art feedback fuzzing that easily scales to large binary applications and unknown environments. We observe that during the execution of a given program, parts of the input often end up directly (i.e., nearly unmodified) in the program state. This *input-to-state correspondence* can be exploited to create a robust method to overcome common fuzzing roadblocks in a highly effective and efficient manner. Our prototype implementation, called REDQUEEN, is able to solve magic bytes and (nested) checksum tests automatically for a given binary executable. Additionally, we show that our techniques outperform various state-of-the-art tools on a wide variety of targets across different privilege levels (kernel-space and userland) with no platform-specific code. REDQUEEN is the first method to find more than 100% of the bugs planted in LAVA-M across all targets. Furthermore, we were able to discover 65 new bugs and obtained 16 CVEs in multiple programs and OS kernel drivers. Finally, our evaluation demonstrates that REDQUEEN is fast, widely applicable and outperforms concurrent approaches by up to three orders of magnitude.

I. INTRODUCTION

Fuzzing has become a critical component in testing the quality of software systems. In the past few years, smarter fuzzing tools have gained significant traction in academic research as well as in industry. Most notably, american fuzzy lop (AFL [44]) has had a significant impact on the security landscape. Due to its ease of use, it is now convenient to more thoroughly test software, which many researchers and developers did. On the academic side, DARPA's Cyber Grand Challenge (CGC) convincingly demonstrated that fuzzing remains highly relevant for the state-of-the-art in bug finding: all teams used this technique to uncover new vulnerabilities.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23371
www.ndss-symposium.org

Following CGC, many new fuzzing methods were presented which introduce novel ideas to find vulnerabilities in an efficient and scalable way (e.g., [10], [16], [19], [31], [34]–[38]).

To ensure the adoption of fuzzing methods in practice, fuzzing should work with a minimum of prior knowledge. Unfortunately, this clashes with two assumptions commonly made for efficiency: (i) the need to start with a good corpus of seed inputs or (ii) to have a generator for the input format. In absence of either element, fuzzers need the ability to *learn* what interesting inputs look like. Feedback-driven fuzzing, a concept popularized by AFL, is able to do so: Interesting inputs which trigger new behavior are saved to produce more testcases, everything else is discarded.

A. Common Fuzzing Roadblocks

To motivate our approach, we first revisit the problem of efficiently uncovering new code, with a focus on overcoming common fuzzing roadblocks. In practice, two common problems in fuzzing are magic numbers and checksum tests. An example for such code can be seen in Listing 1. The first bug can only be found if the first 8 bytes of the input are a specific magic header. To reach the second bug, the input has to contain the string "RQ" and two correct checksums. The probability of randomly creating an input that satisfies these conditions is negligible. Therefore, feedback-driven fuzzers do not produce new coverage and the fuzzing process stalls.

```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
  bug(1);

/* nested checksum example */
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
      bug(2);
```

Listing 1: Roadblocks for feedback-driven fuzzing.

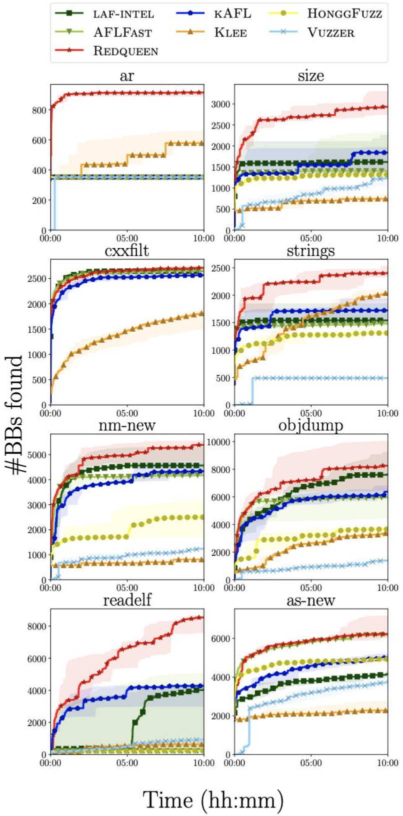
In the past, much attention was paid to address such roadblocks. Different approaches were proposed which typically make use of advanced program analysis techniques, such as taint tracking and symbolic execution [12], [13], [16], [22], [23], [26], [35], [38], [40]. Notably, both ANGORA [16] and T-FUZZ [34] fall into this category. These approaches usually require a rather precise description of the environment (e.g., behavior of library calls or the underlying OS) and the exact semantics of the platform's instruction set. As a result, it is hard to use this methods on targets that use complex instruction set extensions (i. e., floating point instructions) or uncommon

Australian National University DST GROUP DATA 61

A Reproduction Experiment: *RedQueen*

`readelf` results

- *honggfuzz* and *AFLFast* perform poorly
- *RedQueen* is the best performer



Australian National University

DST GROUP

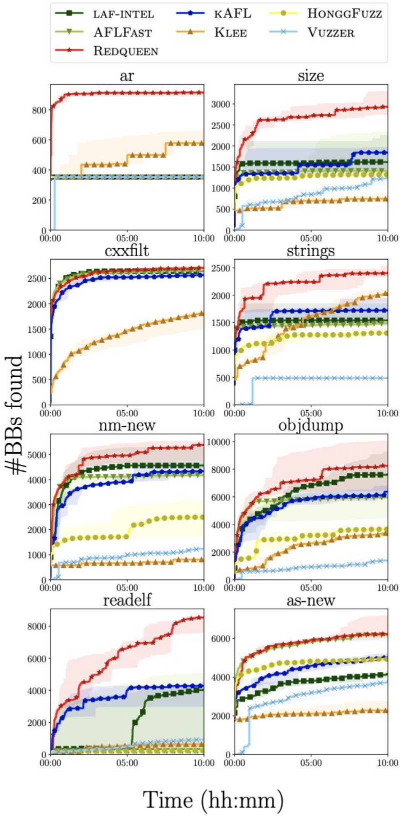
DATA 61

A Reproduction Experiment: *RedQueen*

`readelf` results

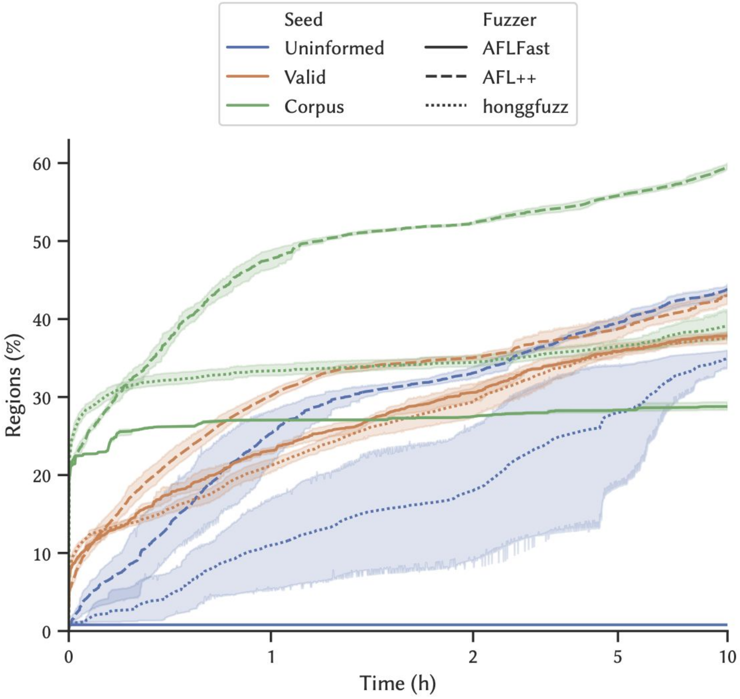
- *honggfuzz* and *AFLFast* perform poorly
- *RedQueen* is the best performer

What if we vary the initial seeds?



Australian National University

A Reproduction Experiment: *RedQueen*



Uninformed	Original ASCII seed
Valid	Singleton ELF (from *AFL*)
Corpus	Collection of ELF files sourced from *AllStar* and *Malpedia* datasets (minimized with `afl-cmin`)

Seed choice matters!

Seed Selection Practices

Since 2018

- 3 studies **do not report seeds**
- 7 studies use **benchmark/fuzzer-provided seeds**
- 2 studies use **manually-constructed seeds**
- 5 studies use **random seeds**
  - 2 studies use a **corpus minimization** tool
- 8 studies use the **empty seed**

Seed Selection Practices

Since 2018

- 3 studies **do not report seeds**
- 7 studies use **benchmark/fuzzer-provided seeds**
- 2 studies use **manually-constructed seeds**
- 5 studies use **random seeds**
    - 2 studies use a **corpus minimization** tool
- 8 studies use the **empty seed**

Corpus Minimization

Why?

- Collecting random seeds may result in behavioral duplication
  - Behaviorally equivalent seeds should be represented by a single seed
- x2 overhead from opening/closing test-cases
  - Minimize size of individual seeds

Corpus Minimization

“Given a large collection of inputs for a particular target (the collection corpus), how do we select a subset of inputs that will form the initial fuzzing corpus?”

Existing Approaches to Corpus Minimization

MinSet

- “Optimizing Seed Selection for Fuzzing”, Rebert et al.
- Models corpus minimization as a **minimum set cover**
- Also weights seeds by **execution time** or **file size**

`afl-cmin`

- Shipped with *AFL*
- Takes into account **edge counts**

Australian National University DST GROUP DATA 61

These approaches are greedy and rely on heuristics

Australian National University

OptiMin

- **Exact** minimum set covers are computable using a **MaxSAT** solver
- Also performs weighted minimizations (file size)
- **6% decrease** in corpus size vs. *MinSet*
- **83% decrease** in corpus size vs. `afl-cmin`

Available at https://github.com/HexHive/fuzzing-seed-selection

Also available in AFL++ at

https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/optimin

Australian National University DST GROUP DATA 61

But what effect does this have on fuzzing?

Australian National University

Evaluation

Benchmarks

- *Magma* (x7 targets)
- *Google Fuzzer Test Suite* (x10 targets)
- "Real-world" programs (x6 targets)

Fuzzers

- *AFL*
- *AFL*++

Corpora

- **FULL** collection corpus
- **EMPTY** seed
- **PROV**ided seeds
- *MinSet* (**MSET**)
- `afl-cmin` (**CMIN**)
- *OptiMin* weighted by file size (**WOPT**)
- WOPT weighted by edge frequencies (**WMOPT**)

Australian National University

DST GROUP

DATA 61

Evaluation

33 CPU-years

Benchmarks

- *Magma* (x7 targets)
- *Google Fuzzer Test Suite* (x10 targets)
- "Real-world" programs (x6 targets)

Fuzzers

- *AFL*
- *AFL++*

Corpora

- **FULL** collection corpus
- **EMPTY** seed
- **PROV**ided seeds
- *MinSet* (**MSET**)
- `afl-cmin` (**CMIN**)
- *OptiMin* weighted by file size (**WOPT**)
- WOPT weighted by edge frequencies (**WMOPT**)

Bug Finding Results

Both AFL and AFL++ perform better when bootstrapped with a minimized corpus, although the exact minimization tool is inconsequential. While both AFL and AFL++ find a similar number of bugs, AFL is generally faster to do so (and with less variance in bug-finding times).

- EMPTY results highly variable, but occasionally the best performer on **highly-unstructured** data (e.g., SoX)
- Low iteration rates + large corpora = negative impact
- x7 CVEs in real-world targets (libtiff, poppler, SoX)

Australian National University

Bug Finding Results

- AFL/AFL++ perform better with one of CMIN, MSET, or W[M]OPT
- AFL generally faster at finding bugs
- EMPTY results highly variable
  - Occasionally the best performer on **highly-unstructured** data (e.g., SoX)
- Low iteration rates + large corpora = negative impact
- x7 CVEs in real-world targets (libtiff, poppler, SoX)

Code Coverage Results

Seed selection has a significant impact on a fuzzer's ability to expand code coverage. When fuzzing with the empty seed, more-advanced fuzzers (e.g., AFL++) are able to cover more code. However, this advantage all but disappears when bootstrapping the fuzzer with a minimized corpus, as faster iteration rates become more critical. The exact minimzation tool remains inconsequential.

- On average, EMPTY explores half as much code
  - Decreases more when mutating highly-structured inputs (e.g., XML)
- Little distinguishes coverage achieved by non-empty corpora (after 18h trial)

Australian National University DST GROUP DATA 61

Code Coverage Results

- AFL/AFL++ perform better with one of CMIN, MSET, or W[M]OPT
- On average, EMPTY explores half as much code
  - Increases when fuzzing with AFL++
  - Decreases more when mutating highly-structured inputs (e.g., XML)
- Little distinguishes coverage achieved by non-empty corpora (after 18h trial)

See our paper for full results

Australian National University

Conclusion

- Choice of fuzzing corpus is a critical and often-overlooked decision
  - It **must** be specified in your paper
- Smarter fuzzers get more mileage out of an empty seed
- Maximize fuzzing yield with minimized corpora

- Code available at https://github.com/HexHive/fuzzing-seed-selection
- Data available at https://datacommons.anu.edu.au/DataCommons/rest/records/anudc:6106/data/

Australian National University