# MoonLight: Effective Fuzzing with Near-Optimal Corpus Distillation


Liam Hayes
*Australian National University*[*]
*& Defence Science and Technology Group*

Hendra Gunadi
*Australian National University*

Adrian Herrera    Jonathon Milford    Shane Magrath
*Defence Science and Technology Group*

Maggi Sebastian
*Australian National University*

Michael Norrish    Antony L. Hosking
*Data61 & Australian National University*





## Abstract

Mutation-based fuzzing typically uses an initial set of valid seed inputs from which to generate new inputs by random mutation. A given corpus of potential seeds will often contain thousands of similar inputs. This lack of diversity can lead to wasted fuzzing effort, as the fuzzer will exhaustively explore mutation from all available seeds. To address this, industrial-strength fuzzers such as American Fuzzy Lop (AFL) come with *distillation* tools (e.g., `afl-cmin`) that automatically select seeds as the smallest subset of a given corpus that triggers the same range of instrumentation data points as the full corpus. Experience suggests that minimizing both the *number* and *cumulative size* of the seeds may lead to more efficient fuzzing, which we explore systematically here.

We present a theoretical foundation for understanding the value of distillation techniques and a new algorithm for minimization based on this theory called MoonLight. The theory allows us to characterize the performance of MoonLight as near-optimal, outperforming existing greedy methods to deliver smaller seed sets. We then compare the effectiveness of MoonLight-distilled seed selection in a long fuzzing campaign, comparing against `afl-cmin`, with MoonLight configured to give weight to different characteristics of the seeds (i.e., unweighted, file size, or execution time), as well as against each target's full corpus and a singleton set containing only an "empty" valid input seed.

Our results demonstrate that fuzzing with seeds selected by MoonLight outperforms the existing greedy `afl-cmin`, and that weighting by file size is usually the best option. We target six common open-source programs, covering seven different file formats, and show that MoonLight outperforms `afl-cmin` in terms of the number of (unique) crashes generated over multiple sustained campaigns. Moreover, MoonLight not only generates at least as many crashes as `afl-cmin`, but also finds bugs that `afl-cmin` does not. Our crash analysis for one target reveals four previously unreported bugs, all of which are security-relevant and have received CVEs. Three of these four were found only with MoonLight.


[*]This work performed as an ANU summer scholar.

## 1 Introduction

Fuzzing is a dynamic analysis technique for finding bugs and vulnerabilities in software, aiming to trigger 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. A given corpus of potential seeds will often contain thousands of inputs that generate similar behavior in the target, which can lead to wasted fuzzing effort, as the fuzzer will exhaustively explore mutation from all available seeds.

Due to their simplicity and ease of use, mutation-based fuzzers such as AFL [32], honggfuzz [27], and libFuzzer [26] are widely deployed in industry, where they have been highly successful in uncovering thousands of bugs across a large number of popular programs [4]. This success has prompted much research into improving various aspects of the fuzzing process, including mutation strategies [17], seed selection policies [9], and path-exploration algorithms [31].

In addition, current research often cites the importance of high-quality input seeds and their impact on fuzzer performance [15, 22, 23, 29]. While the importance of seed selection seems to be well-known [15], comparatively little research [22, 23] has focused on the problem of *optimal design and construction of corpora* for mutation-based fuzzers. Intuitively, there are several properties one might desire of the collection of seeds that form the initial corpus:

**Property 1** (Maximize coverage of target behaviors)**.** *The seeds in the corpus should collectively span the range of observable behaviors of the target. Fuzzers typically approximate this with code coverage, so seeds in the corpus should collectively execute as much of the code as possible. A lack of coverage diversity in seeds inhibits coverage testing during a fuzzing campaign.*

**Property 2** (Eliminate redundancy in behavior across seeds)**.** *Seeds that are behaviorally similar to one another (following from property 1: that produce the same code coverage) should* *be represented in the corpus by a single seed. Fuzzing multiple seeds with the same behavior is wasteful [23].*

**Property 3** (Minimize the total size of the corpus)**.** *This reduces storage costs and results in a polynomial reduction in the mutation search space (described later).*

**Property 4** (Minimize the sizes of the seeds)**.** *Contention in the storage system should be avoided where possible. Fuzzers are highly I/O bound [30], so smaller seed files should be preferred to reduce I/O requests to the storage system. In turn, this will shorten the execution time of each iteration, achieving more coverage in any fixed amount of time.*

Under these assumptions, simply gathering as many input files as possible is not an optimum approach for constructing a fuzzing corpus (due to properties 2 to 4 above). At the other extreme, our assumptions also suggest that beginning with the "empty corpus" (consisting of one zero-length file, say) would be less than ideal (due to property 1).

The following natural questions arise: (i) How do we best select seeds for a fuzzing corpus? (ii) If we assume properties 1 to 4 above, how should those assumptions be weighted with respect to each other? (iii) Having generated answers to our first two questions, does the resulting approach produce corpora that in turn produce better results (more bugs found in the same amount of time) than alternative, state-of-the-art approaches?

We call the process of seed selection *corpus distillation*.[1] We assume that we already have a large candidate corpus, in the form of inputs already gathered, and so we can also explore distillation strategies that range from throwing away all the seeds entirely through to keeping all the candidate seeds. In this way, we can test our assumptions above.

**Contributions.** We propose a new method for corpus distillation that reliably performs at least as well as AFL's `afl-cmin` tool, which represents current available state-of-the-art:

- We have designed and implemented an open-source framework, *MoonLight*,[2] that represents the distillation problem as a *(weighted) minimum set cover problem* (WMSCP) and efficiently computes a solution using a dynamic programming approach. MoonLight extends Rebert *et al.*'s work [23] by also *quantifying the upper-bound divergence* from an optimum solution (Section 3).
- We perform extensive empirical evaluation of MoonLight by: (i) comparing its performance in terms of reducing corpus size against a typical greedy algorithm (Section 3.4), and (ii) testing crash and bug-finding outcomes in a fuzzing campaign on six popular open-source targets, testing MoonLight-distilled corpora against the extreme points of our spectrum (a full corpus, and the empty corpus), as well against the state-of-the-art `afl-cmin` tool (Section 4).
- Our evaluation shows that, up to statistical significance, and in terms of numbers of crashes generated over campaigns, MoonLight outperforms the competing approaches: it is never worse, and is occasionally better.
- The ultimate aim of fuzzing is to uncover bugs in software. To this end, we have triaged crashes, and determined that MoonLight not only generates at least as many crashes, but it is also capable of finding bugs that other approaches do not. Analysing crashes for one target we found seven bugs, of which four are security-interesting. Indeed, three of these bugs were only found by our tool. We have logged bug reports and received CVEs for these bugs.

[1] Distillation might also be referred to as *reduction* or *minimization*. We choose to use the same language as Pailoor *et al.* [22], and avoid the term *reduction* since it is also used in the crash triage process to reduce crash exemplars to a minimum size [12, 33].

[2] Available at `https://bit.ly/2X5scV8`

## 2 Background and Related Work

We begin with a brief summary of different fuzzing approaches and the literature describing them. We then focus on the problem of corpus distillation, formalizing the problem as it applies to mutation-based fuzzers, and then discussing existing corpus distillation techniques.

### 2.1 Fuzzing

Fuzzing has become a popular technique for automatically finding bugs and vulnerabilities in software. This popularity can be attributed to fuzzing's simplicity and its success in finding bugs in "real-world" software [4, 26, 27, 32]. At a high level, fuzzing involves the generation of large numbers of test-cases that are fed into the target program to induce a crash. The target is monitored so that crash-inducing test-cases can be identified and saved for further analysis/triage after the fuzzing campaign has ended.

How a fuzzer generates test-cases depends on whether it is *generation*-based or *mutation*-based. Generation-based fuzzers (e.g., QuickFuzz [11], Dharma [20], and CodeAlchemist [13]) require a specification/model of the input format. This specification is then used to synthesize test-cases. In contrast, mutation-based fuzzers (e.g., AFL [32], honggfuzz [27], and libFuzzer [26]) require an initial corpus of seed inputs (e.g., files, network packets, environment variable strings, etc.) to bootstrap test-case generation. New test-cases are then generated by mutating inputs in this initial corpus.

Perhaps the most popular mutation-based fuzzer is American Fuzzy Lop (AFL) [32]. AFL is a *greybox* fuzzer, meaning that it uses light-weight instrumentation to gather *code coverage* information during the fuzzing process. This code coverage information acts as an approximation of program behavior. AFL instruments edge transitions between basic blocks and uses this information as code coverage. By feeding the code coverage information back into the test-case generation algorithm, the fuzzer is able to explore new executions (and hence behaviors) in the target. In addition to the core fuzzer, AFL also provides a corpus distillation tool: `afl-cmin`. `afl-cmin` will be discussed in more detail in Section 2.3, after we formalize the corpus distillation problem.

### 2.2 Formalizing the Distillation Problem

Our work focuses on solving the problem of optimal design and construction of corpora for mutation-based fuzzers. To solve this problem, the primary question we need to answer (as posed by Rebert *et al.* [23]) is:

**Question:** 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*?

We refer to the process of selecting this subset of inputs as *distillation*. In particular, we are most interested in distillation that leads to *more efficient fuzzing*. As the ultimate aim of fuzzing is to uncover bugs in software, this means producing a *higher crash yield* than if we had simply used the collection corpus as the fuzzing corpus. This is because most seeds in a collection corpus are behaviorally very similar to each other. Therefore it is important to distill the possibly very large collection corpus into a much smaller fuzzing corpus, which is the minimum set of seeds that still spans the whole set of observed program behavior.

Previous work has formalized distillation as an instance of the *minimum set cover problem* [1, 2, 23]. The minimum set cover problem (MSCP) states that given a set of elements $U$ (the universe) and a collection of $n$ sets $S = \{s_1, s_2, ..., s_n\}$ whose union equals the universe $U$, what is the *smallest* subset of $S$ whose union still equals the universe $U$. This smallest subset $\mathbb{C} \subseteq S$ is known as the *minimum cover set* or the *minset*. Note that there may be many different minimum cover sets, all of the same minimum size.

Each element in $S$ is often associated with some cost (or weight), $c$. In this case, the *weighted* MSCP (WMSCP) attempts to minimize the total cost of the elements in $\mathbb{C}$. Note that the MSCP is equivalent to the WMSCP when $c = 1$ for all $n$ elements in $S$.

The (W)MSCP is NP-complete [14] and a *greedy* algorithm is typically used to find an approximate solution [5]. While it is possible to use a greedy algorithm and maintain a performance bound on solution quality [5], in practice this bound is very large and not very useful.

Corpus distillation can thus be formalized as a (W)MSCP, where $S$ is a set of $n$ seeds from a collection corpus, and each element in $S$ encodes code coverage information for that particular seed. Code coverage is conventionally used to characterize seeds in $S$ due to the strong positive correlation between code coverage and bugs found while fuzzing [10, 16, 19, 21]. Finding $\mathbb{C}$ is, therefore, equivalent to finding the minimum set of seeds that still maintains the code coverage observed in the original collection corpus. By definition, $\mathbb{C}$ satisfies properties 1 to 3 listed in Section 1. By solving a WMSCP, where the weights are based on individual seed sizes, we can also satisfy property 4.

### 2.3 Existing Approaches to Distillation

The distillation problem in fuzzing has been tackled in a number of different ways over the years.

Abdelnur *et al.* [1] introduced the idea of computing $\mathbb{C}$ over code coverage as a seed selection strategy. They used a simple greedy algorithm to solve the unweighted MSCP.

Rebert *et al.* [23] extended that work by also computing $\mathbb{C}$ weighted by execution time and file size. They designed six corpus distillation techniques and empirically evaluated these techniques over a number of fuzzing campaigns. Rebert *et al.*'s findings, given their experiments, were that an unweighted greedy-reduced distillation performed best in terms of distillation ability, and that the PEACH SET algorithm (based on the Peach fuzzer's `peachminset` tool [6], described later) found the highest number of bugs. Our paper extends Rebert's work with a new theory and experimental design.

MoonShine [22] is a corpus distillation tool for OS fuzzers. OS fuzzers typically test the system-call interface between the OS kernel and user-space applications. As such, the seeds that are distilled by MoonShine are a list of system calls gathered from program traces. In contrast, our work targets file-format fuzzing, which is a fundamentally different problem to distilling system calls, and requires a vastly different approach.

SmartSeed [18] takes a different approach to those previously described. Rather than distilling a corpus of seeds, SmartSeed instead uses a machine learning model to generate "valuable" seeds, where a seed is considered valuable if it uncovers new code or produces a crash.

As previously mentioned, the Peach fuzzer [6] ships with its own corpus distillation tool, `peachminset`. Despite its name, Rebert *et al.* [23] showed that it does not in fact calculate the minimum cover set nor a proven competitive approximation thereof.

Due to AFL's popularity, `afl-cmin` is perhaps the most widely-used corpus distillation tool. `afl-cmin` implements a greedy distillation algorithm but has a unique approach to coverage. `afl-cmin` reuses AFL's edge coverage to categorize seeds at distillation time. `afl-cmin` chooses the smallest seed in the collection corpus that covers a given edge and then performs a greedy weighted reduction. We consider `afl-cmin` to be the state-of-the-art in corpus distillation tools and use it as our baseline when evaluating our tool MoonLight.

## 3 Our Approach

In this section we provide an intuitive description of our corpus distillation algorithm. A more detailed presentation of the theory and key accompanying proofs are provided in Appendix A.1.

**Problem Definition.** Recall that our fundamental task is *distillation*: the removal of members of a set of seeds (a *corpus*) while keeping the *coverage* of the corpus unchanged. Our target has a finite set of edges between its constituent basic blocks, and any given seed will cause a subset of these edges to be traversed when the target is executed on that seed. This seed's set of edges is its *coverage*, and the coverage of a corpus is the union of all that set's coverages.

Following current state-of-the-art fuzzers (in particular, AFL), we make the assumption that edge coverage is a good approximation of target behavior, and thus: fuzzing over a distilled corpus will discover as many crashes as fuzzing over the original corpus. To the extent that this assumption is not true, we are willing to forgo lost behaviors (which may lead to crashes) in exchange for increased testing speed in the fuzzing process. Pragmatically, we hope that this increased speed will result in the discovery of more crashes. Empirically, this hope is indeed borne out and discussed in Section 4.

The coverage data for a corpus can be viewed as a matrix: each row corresponds to one seed, and each column to a possible edge between basic blocks in the target. Such a matrix $A$ has $A_{ij} = 1$ if seed $i$ causes the target to traverse edge $j$, and is zero otherwise.

Our algorithm, implemented in the MoonLight tool, applies dynamic programming to take a large coverage matrix and recursively transform it through row and column eliminations into successively smaller matrices whilst accumulating a minimum cover set $\mathbb{C}$. As discussed in Section 2.2, $\mathbb{C}$ is the minimum collection of seeds that still spans all the observed edges in the target when dynamically traced. This minimum cover set becomes our *distilled* fuzzing corpus to be used instead of the original collection corpus.

We use the following notation and specify several matrix operations to explain our method:

- $S$ is the set of all *seeds* in the full corpus;
- $E$ is the set of all *edges* executed by the target for each element in $S$;
- $N = \|S\|$ is the number of seeds in the full corpus;
- $M = \|E\|$ is the number of edges in the target;
- $s_i \in S$ is the $i$th seed under some ordering;
- $e_j \in E$ is the $j$th edge under some ordering;
- $A = \left[a_{i,j}\right]$ is the $N \times M$ corpus **coverage matrix**; and
- Let $c_i$ be the *weight* of seed $s_i$—a value to be defined but often 1 or the *file size* of the seed or its execution time.

### 3.1 Problem

We formulate our problem as a WMSCP described as follows:

**OBJECTIVE**:

$$\min \sum_{i=1}^{N} c_i x_i$$

**SUBJECT TO:**

$$\sum_{i=1}^{N} a_{ij} x_i \geq 1, \forall j \in \{1, \ldots, M\}$$

and

$$x_i \in \{0, 1\}, \forall i \in \{1, \ldots, N\}$$

The objective is to find the smallest weighted set of seeds that still covers all the columns (edges) in the matrix. An important point to note is that this formulation assumes that the *column sums are non-zero*. We term column sums that equal zero *singularities*. Singularities relate to edges that have not been traversed and hence can be safety removed from the matrix.

The unweighted version of the problem (the MSCP, as defined in Section 2.2) sets $c_i = 1, \forall i = \{1, \ldots, N\}$. This simply finds the smallest set of rows (i.e., the number of seeds) that spans all of the columns. In the following presentation we generally do not distinguish between the weighted and unweighted formulations except where the distinction is important to the computation.

### 3.2 Matrix Operations

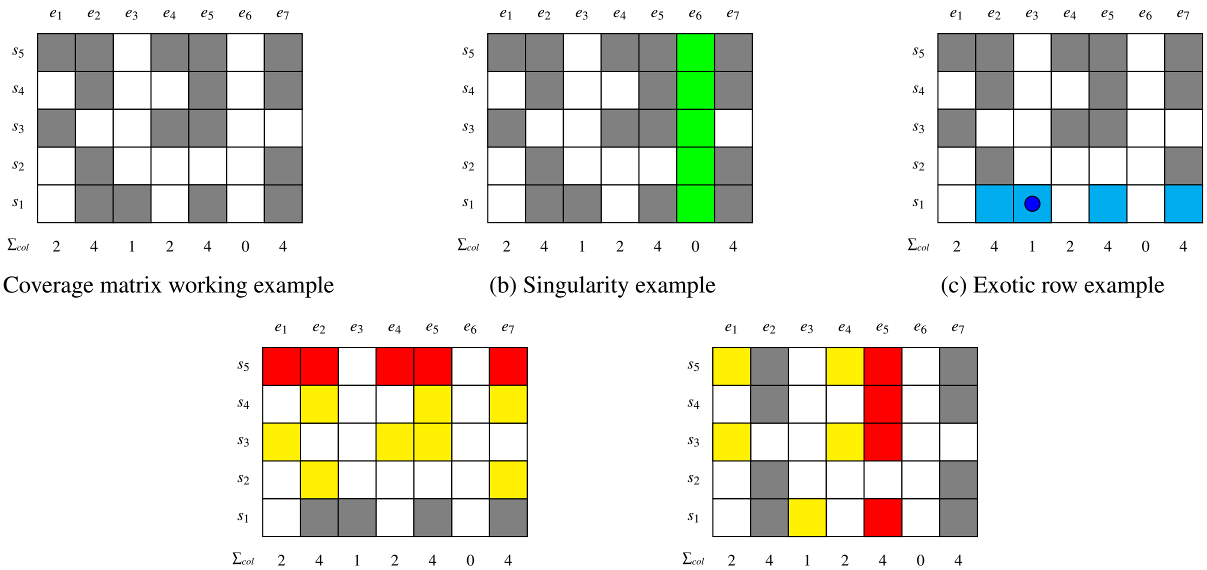
The corpus coverage matrix $A$ is typically a large and sparse matrix. For example, the number of seeds $N$ in the corpus can be of the order $10^5$ files, while the number of observed edges is of the order $10^6$. Combined, the coverage matrix is in the order of $10^{11}$ elements.

Before presenting the algorithm we need to define our *operational primitives* which allow us to transform the large original problem into smaller over-lapping sub-problems. Here we present an intuitive description of each of the operations: a more formal presentation is made in the appendix.

We proceed with the working example in Fig. 1a to describe our algorithmic operations.

**Singularities.** We define the term *singularity* to describe the situation where a column or row sum is *zero*. Row sum singularities are rare in practice and are pathological since they represent seeds that do not cover any code when parsed by the target. These seeds can simply be ignored. In contrast, column singularities are very common. Column singularities tend to be an artifact of the tracing tool which presumably identified *all* edges in the target.

Consider Fig. 1b from the working example. Column $e_6$—shown in green—is a singular column. We can eliminate this column to produce a smaller matrix $A'$ whose minimum cover set $\mathbb{C}$ is the same as the original matrix $A$.



(a) Coverage matrix working example

(b) Singularity example

(c) Exotic row example

(d) Row dominance example

(e) Column dominance example

Figure 1: Coverage minimization examples. Rows $s_i$ represent seeds; columns $e_i$ represent all possible edges. Gray elements indicate edges that are covered by a given seed, while white elements indicate those edges that are uncovered.

**Exotic Rows.** An *exotic column* $e^*$ in $A$ has a column sum of *one*. An *exotic row* is the unique row $s^*$ in $A$ that covers $e^*$. For example in Fig. 1c the row $s_1$—shown in blue—is exotic because it is the only row that covers column $e_3$. All seeds associated with exotic rows are by definition a part of the final solution and will be included in the distilled corpus.

**Dominant Rows.** Row dominance captures the intuitive idea that some rows in $A$ may be a subset of a single row. The larger row *dominates* the smaller *submissive* row which is a subset of the *dominator*. A dominant row may in turn be dominated. A dominant row with no dominator we call *primal dominant*. Intuitively, all submissive rows can be deleted from the coverage matrix $A$. However, this operation is row weight sensitive. If the submissive row has a larger row weight than the dominator then it can be deleted else it is left alone because it may ultimately lead to smaller weighted corpus.

Continuing with the (unweighted) working example in Fig. 1d we identify row $s_5$ as a dominant row—shown in red—since it dominates rows $S = \{s_2, s_3, s_4\}$—shown in yellow. The three submissive rows can be deleted from the matrix.

**Dominant Columns.** Column dominance is a similar idea to row dominance. In this case some columns in $A$ may be a subset of a single column. However, in this operation the dominated column is deleted and the submissive columns are left alone. The reasoning here is that any final solution by definition will contain seeds that cover the submissive columns and by implication they will also cover the dominant column. The dominant column is redundant and can be safely removed.

Continuing with the working example in Fig. 1e, we identify column $e_5$ as a dominant column since it dominates columns $E = \{e_1, e_3, e_4\}$. Note that $e_2$ is not dominated by $e_5$ because of $s_2$.

**Contained Columns.** When choosing row $s_i$ to add to our solution we can eliminate all the columns that $s_i$ covers. We call this operation *contained column eliminations*. The columns can be safely deleted because they will be covered by the seed associated with row $s_i$. This is a very powerful technique for reducing the dimensionality of the problem.

**Heuristic Row Reduction.** Up until now the row and column eliminations that we have described have been optimal in the sense that we have been able to guarantee that an optimal solution for a smaller transformed matrix can be used to construct an optimal solution for the larger matrix. However, there are times when all optimal transformations are exhausted and you must make a *heuristic choice* to select a row to add to $\mathbb{C}$. Intuitively, a good heuristic is to select the row with the largest *rowsum*, since this will trigger a large *contained columns elimination*. In the weighted case we choose the row with with the largest *rowsum*/*rowweight*. The expectation is that optimal transformations can be resumed on the smaller matrix. We quantify our divergence from an optimal solution in Section 3.3.1.

## 3.3 The MoonLight Algorithm

We now present the MoonLight algorithm. MoonLight uses dynamic programming to accumulate a near optimal set cover through successively applying the matrix transformations that we described above. The strategy is to perform transformations until the final transformation results in the Null matrix $Z$ and no further operations are possible.

### 3.3.1 Dynamic Programming Formulation

Given the set of matrix reduction operators just presented we use a dynamic programming formulation of the problem to compute a near optimal sequence of operations to construct $\mathbb{C}$. Dynamic programming analysis begins with identifying the cost function of a state (coverage matrix) in terms of a transition cost for using an operator and the subsequent cost of the new state smaller state. This recursive function gives the so called Bellman equation [3] for the problem.

We define the *state* of the problem to be specified by the coverage matrix $A$. We define the *cost function* of the state to measure the divergence from an optimal cover set. If the cost function for a state is computed to be zero then by implication an optimal cover set was computed. If the cost is non-zero then this cost measures the divergence from an optimal cover set. The cost of the state is defined to be the cost of any matrix operation performed on $A$, denoted by $d_\star(A,A')$, *plus* the cost of the subsequent state $A'$. Note that the cost function does not involve a measurement of the cover set.

Clearly, we want to choose the optimum transformation for state $A$. Therefore, we define the Bellman equation for the problem as:

$$Cost(A) = \min_{i \in \text{operators}} \begin{cases} d_i(A,A') + Cost(A') & A \neq Z \\ 0 & A = Z \end{cases} \quad (1)$$

Where the indexation consists of the five matrix operations previously specified and $Z$ is our terminating condition. Each of the operators produce a set of rows and columns which are removed during the transition from $A$ to $A'$.

What are the costs $d_\star$ for each operation? Any optimal row or column elimination operation has zero cost because they contribute to an optimal cover set computation. The objective is to find an *optimum* set cover if possible or measure divergence from optimality when necessary. Optimal reductions occur when $minset(A) = minset(A') \bigcup ExoticRow$. We specify these reductions to have a zero cost *iff* the matrix size is reduced (i.e., $A \neq A'$), else they have an infinite cost if they make the matrix larger. For example, removing column singularities and selecting exotic rows are all optimal operations and incur no cost providing they reduce the size of the matrix $A'$. In contrast, heuristic row reductions cannot be guaranteed to be an optimal reduction. Therefore, we define the dynamic programming cost of this operation to be: (i) *one* for unweighted set cover; or (ii) $rowsum/rowweight$ for weighted set cover.

The reasoning is this: if by chance the heuristic row choice was optimal then before and after minsets are the same. If it was not optimal then the before and after minsets deviate by one seed. This cost represents an upper bound measure of divergence from the size of the optimal choice at state $A$. That is $|minset(A)| - |minset(A')| \leq 1$.

Consider the worst case, where the heuristic row choice, $x_h$ is not part of an optimal solution. An optimal solution to $A$ is also a solution of the reduced matrix $A'$, since the columns of $A'$ are a subset of the columns of $A$. Hence, $Cost(A') \leq Cost(A)$. At worst, the inclusion of row $x_h$ increases the cost of the solution by $c_{x_h}$, but reduces the cost of an optimal solution by nothing. Hence $c_{x_h}$ represents the upper bound divergence from the size of the optimal choice and is the operation cost of the heuristic reduction.

### 3.3.2 MoonLight Algorithm

We present MoonLight in Algorithm 1. The ordering of the five basic matrix operations reflects the relative importance of those operations in either dimensional reduction, selecting necessary rows, or computational complexity. Given that matrix operational costs for $d_\star$ are zero apart from heuristic choice, this formulation of the algorithm is a correct encoding of the Bellman Equation outlined in Section 3.3.1. Observe that the heuristic row choice is the last resort option.

Algorithm 2 presents the Greedy algorithm. As can be observed, it is simply a continuous iteration of heuristic row choices. That is, for unweighted distillations it is the largest *rowsum* and for weighted distillations it is the largest $rowsum/rowweight$.

### 3.3.3 Implementation

We implemented MoonLight as a standalone tool in approximately 2000 lines of C++. The MoonLight tool takes as input a directory containing (i) the seed files to distill; and (ii) for each seed file, a *compressed* code coverage trace.

Because the number of seeds can be quite large, we store code coverage as a *bit vector* to reduce both storage costs and memory costs when computing $minset(A)$. The index $i$ of each bit in the bit vector corresponds to a specific edge in the target. Moreover, the $i$-th bit in every trace file in the corpus corresponds to the exact same $i$-th edge in the target.

In addition, we provide a tool, *MoonBeam*, to generate these trace files. MoonBeam is implemented in approx. 160 lines of Python. MoonBeam converts the output of `afl-showmap` (a tool included with AFL to display the coverage trace of a particular input) to the bit vector representation previously described. Note that the output of `afl-showmap` is also used by `afl-cmin`.

Both MoonLight and MoonBeam have been open-sourced and are available at https://bit.ly/2WZVynP.

## 3.4 Preliminary Results

We present a brief preliminary study of the performance of the MoonLight-derived corpus compared to the commonly used

**Algorithm 1** MoonLight

```
function MoonLight(A, X)
    if isEmpty(A) then
        return X
    end if
    Cols ← Singularities(A)                          ▷ Singularities
    if Cols ≠ ∅ then
        A′ ← RemoveRowsCols(A, ∅, Cols)
        return MoonLight(A′, X)
    end if
    Rows ← Exotic(A)                                 ▷ Exotic Rows
    if Rows ≠ ∅ then
        Cols ← ContainedCols(A, Rows)
        A′ ← RemoveRowsCols(A, Rows, Cols)
        X′ ← X ⋃ Rows
        return MoonLight(A′, X′)
    end if
                                                     ▷ Primal Dominant Rows
    (DomRows, SubRows) ← DominantsRows(A)
    if DomRows ≠ ∅ then
        Cols ← ContainedCols(A, DomRows)
        DelRows ← DomRows ⋃ SubRows
        A′ ← RemoveRowsCols(A, DelRows, Cols)
        X′ ← X ⋃ DomRows
        return MoonLight(A′, X′)
    end if
                                                     ▷ Primal Dominant Columns
    Cols ← DominantCols(A)
    if Cols ≠ ∅ then
        A′ ← RemoveRowsCols(A, ∅, Cols)
        return MoonLight(A′, X)
    end if
    Rows ← Heuristic(A)                              ▷ Heuristic Row
    Cols ← ContainedCols(A, Rows)
    A′ ← RemoveRowsCols(A, Rows, Cols)
    X′ ← X ⋃ Rows
    return MoonLight(A′, X′)
end function
Solution ← MoonLight(A, ∅)                           ▷ Main
```

Greedy distilled corpus. We distill five different file types (PDF, DOC, PNG, TTF, and HTML) and present the results in Table 1. The corpora consist of proprietary sourced seeds which are unable to be released. The corpus data in Section 4 is not proprietary.

With reference to Table 1, it can be seen that the MoonLight derived corpus always results in a smaller fuzzing corpus than a Greedy distilled corpus, for both weighted and unweighted scenarios. Moreover, in Table 1 we also see that the MoonLight-derived corpus is also at most one seed away from an optimum sized corpus. It is important to note that the reason our MoonLight algorithm works well on corpus data is because the coverage matrix is not random. There are strong

**Algorithm 2** Greedy

```
function Greedy(A)
    X ← ∅
    while A ≠ Z do
        Rows ← Heuristic(A)                          ▷ Heuristic Row
        Cols ← ContainedColumns(A, Rows)
        A ← RemoveRowsCols(A, Rows, Cols)
        X′ ← X ⋃ Rows
    end while
    return X
end function
Solution ← Greedy(A)                                 ▷ Main
```

correlations between edge coverage measurements. When tested on random matrices our algorithm performs the same as Greedy.

These results demonstrate the efficacy of the tool and our approach. However, they say nothing about the effectiveness when applied to fuzzing. Therefore, we require an experimental evaluation to decide if MoonLight distilled corpora result in better fuzzing outcomes. This is addressed in the remainder of this paper.

#### 3.4.1 Motivating Weighted Corpus Distillations

We expect there to be several benefits of using *weighted* corpus distillations in fuzzing where the weights are *file sizes* rather than other measures (e.g., execution times). The first benefit is a significant reduction in the search space. The second benefit is in the engineering effort required.

**Search space reduction.** Consider a file $X$ that is $N$ bytes long with coverage $C$. Now consider a second file $Y$ that is $M$ bytes long with the same coverage $C$.

The number of byte-limited mutations for file $X$ is $s = 2^N$ possibilities. Similarly, the number of byte-limited mutations for file $Y$ is $t = 2^M$ possibilities. That is $t = s^{\frac{M}{N}}$. Given this exponential relationship in mutation possibilities in file sizes there is a strong incentive in mutational fuzzing to find the smallest set of *small* files that spans the set of observed edges. Doing so results in a *significant* reduction of *inadvertent and unintended duplicate mutational fuzz tests*.

**Engineering.** Industrial scale fuzzing involves a large number of worker processes (typically thousands) campaigning on a given target. A common architecture in this scenario is to serve files from a centralized network attached storage device. The result is that the performance of industrial scale fuzzers are often I/O constrained since very high iteration rates demanded of each worker results in even higher I/O operations per second to the central storage. Therefore, a weighted corpus distillation, because it minimizes the total collective byte size of the fuzzing corpus, alleviates the I/O demand on the storage.

Table 1: Preliminary distillation results. Each corpus is summarized by its number of files (#) and its weight (GB). A -U suffix indicates that the result is due to an *unweighted* distillation, while a -S suffix indicates that the result is due to a distillation *weighted by file size*.

| File type | Collection # | Optimal # | MoonLight-U # | Greedy-U # | Greedy-S # | MoonLight-S # | Greedy weight (GB) | MoonLight weight (GB) |
|---|---|---|---|---|---|---|---|---|
| PDF | 42,056 | 663 | 664 | 727 | 1240 | 855 | 648,260 | 606,841 |
| DOC | 7,836 | 744 | 745 | 755 | 1009 | 777 | 1,126,046 | 1,071,298 |
| PNG | 4,831 | 94 | 94 | 145 | 138 | 104 | 13,730 | 13,251 |
| TTF | 5,666 | 27 | 27 | 28 | 31 | 27 | 1,156 | 1,121 |
| HTML | 69,991 | 409 | 410 | 581 | 756 | 530 | 90,062 | 84,925 |

Given these two performance considerations practical fuzzing would seem to benefit most from using file-size weighted distillations compared to unweighted.

## 4 Evaluation

This section presents an evaluation of six different corpus design approaches which we soon describe in detail. One technique, MoonLight weighted by file size, is shown to statistically outperform all other design techniques. However, there are a range of other important findings we will present based on the large number of experiments we have conducted.

### 4.1 Methodology

**Experimental setup.** We used two machines in our experiments. Both machines have 48-core Intel(R) Xeon(R) Gold 5118 2.30GHz CPUs with 512GB of RAM and Hyper-Threading enabled (providing a total of 96 logical CPUs). One machine runs a hypervisor (the virtual machine runs Ubuntu 16.04), while the other machine runs Ubuntu 18.04 natively. We conducted all experiments for a particular target on the same machine.

**Sample collection.** We selected six popular open-source programs and seven different file formats to test different corpus design techniques. These targets are detailed in Table 2. The driver program used for each library is shown in brackets.[3] The benchmarks were selected to be representative of real-world programs while also operating on a diverse range of file formats (e.g., images, audio, documents, etc.).

For each file type, we built a Scrapy-based[4] crawler to crawl the Internet for 72 hours to create the collection corpus. For image files, crawling started with Google results and the Wikimedia Commons repository. For media files and document files (e.g., PDF), crawling started from the Internet Archive and Creative Commons collections. Large files (greater than 300KB) were removed to satisfy AFL's usage recommendations. We found TIFF files to be relatively rare, so 40% of the TIFF seeds were generated by converting other image types such as JPEG and BMP using ImageMagick.

Table 2: Fuzzing targets

| Program (driver) | Version | File type |
|---|---|---|
| Poppler (`pdftotext`) | 0.64.0 | PDF |
| SoX (`sox`) | 14.4.2 | MP3 |
| SoX (`sox`) | 14.4.2 | WAV |
| librsvg (`rsvg-convert`) | 2.40.20 | SVG |
| libtiff (`tiff2pdf`) | 4.0.9 | TIFF |
| FreeType (`char2svg`) | 2.5.3 | TTF |
| libxml2 (`xmllint`) | 2.9.0 | XML |

Each collection corpus was preprocessed to remove duplicate files, and files larger than 300KB.Duplicate files were found by computing an MD5 sum of each file. The cutoff file size 300KB was chosen as a best effort to conform to AFL author's suggestion about seed's file size, while still having enough eligible seeds in the preprocessed corpora. Audio files larger than 1MB were divided into smaller files using FFmpeg. In total, we collected 2,513,175 seeds across seven different file formats. After preprocessing our collection corpus we were left with a total of 524,891 seeds.

**Fuzzer setup.** Each fuzzing campaign (i.e., each fuzzing trial per target/file-type) was conducted for 18 hours *and repeated 30 times*. We have emphasized the large number of repeated trials here because we found, consistent with Klees *et al.* [15], that individual fuzzing campaigns are wildly variant in performance and, therefore, reaching statistically sound conclusions requires at least this number of trials in practice. The length of each campaign and the number of times that they were repeated satisfy the recommendations presented by Klees *et al.* [15].

AFL was configured for single-system parallelization[5] with one master and seven secondary nodes. Each target was com-

[3] `char2svg` was adapted from https://www.freetype.org/freetype2/docs/tutorial/example5.cpp.
[4] https://scrapy.org/

[5] https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt

piled using AFL's LLVM-based instrumentation with Address Sanitizer (ASan) [25] enabled. The LLVM-based instrumentation was chosen over AFL's assembler-based instrumentation because it offers the best level of interoperability with ASan.

The amount of virtual memory made available to each target varied and required some tuning for effective use. All other parameters were left at their default values.

**Experiment.** We evaluate six distillation techniques (see below) against the seven targets shown in Table 2. For each distillation technique we performed 30 distinct trials of 18 hours of fuzzing each with the same distilled corpus, using the latest version of AFL (2.52b). In total this amounts to 1,260 individual trials and 181,440 hours of fuzzing.

We evaluate the following six distillation techniques:

**Full** The collection corpus without applying any distillation other than the previously discussed duplicate removal and size filtering.

**`afl-cmin`** AFL's accompanying tool for corpus distillation as described in Section 2.3.

**MoonLight-U** The MoonLight *unweighted* algorithm, as presented in Section 3.

**MoonLight-S** The MoonLight algorithm *weighted by file size*.

**MoonLight-T** The MoonLight algorithm *weighted by execution time*.

**"Empty" seed** Unlike Rebert *et al.* [23], we also consider a corpus consisting of just the "empty" seed. This follows the findings of Klees *et al.* [15], which *"despite its use contravening conventional wisdom"*, showed the empty seed outperforming (in terms of crash yield) a set of valid, non-empty seeds for some targets.

In fact, for some targets we did not use a zero-sized seed but rather a small file manually crafted to contain only the bytes necessary to satisfy file header checks. More details on these "empty" seeds can be found in Appendix A.4.

**Statistical tests.** Current fuzzing research often lacks a statistically appropriate experimental method that accounts for the very high levels of variance found between individual fuzzing trials. Attention was recently drawn to this by Klees *et al.* [15] and we follow their recommendations.

We employ three statistical tests: the first test (Mann-Whitney U-test) decides if the *difference* between two (A/B) empirical population distributions is significant (measured by p-value). In our experiments we use `afl-cmin` as the benchmark for the B population and only accept significance when p-values are less than 0.05. We benchmark against `afl-cmin` because neither Rebert *et al.*'s tool [23] nor SmartSeed [18] are publicly available. The second test, the Vargha-Delaney A measure [28], decides how strong that difference is. An A/B experiment can be statistically different when measured by p-values but in fact the effect size of the difference may still be weak. Effect sizes are classified into *negligible*, *small*, *medium*, and *large* (as defined by Romano *et al.* [24]). The third test is the "coefficient of variation", which is simply the standard deviation divided by the mean of a distribution. It allows us to rank and compare different similar random processes: we prefer distributions with high means and low variance.

We use these statistics on two sources of experimental data:

**Crash yield** The number of unique crashes found during an individual trial (fuzzing run);

**AUC** An area under the curve (AUC) measurement that summarizes the fuzzer response over time (iterations). The fuzzer response is the step-wise plot of cumulative unique crashes against iteration (test number). Fig. 2 shows our fuzzer response curves.

AUC measurements are important because they *"reflect that finding crashes earlier and over time is preferred to finding a late burst"* [15].

However, what ultimately matters is the effectiveness of an approach at finding security-interesting bugs. To this end, we triage the crashes for one target, isolating the bugs that led to those crashes, and analyze how effective `afl-cmin` and our new approach are at finding those bugs.

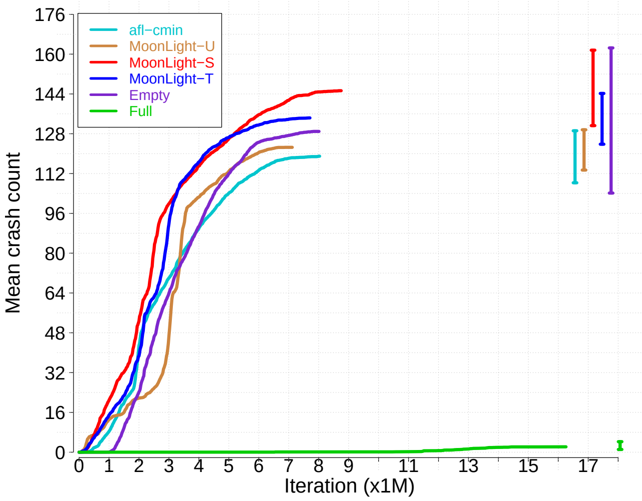
## 4.2 Experimental Results

Our experimental results are summarized in Fig. 2 and Tables 3 to 5.

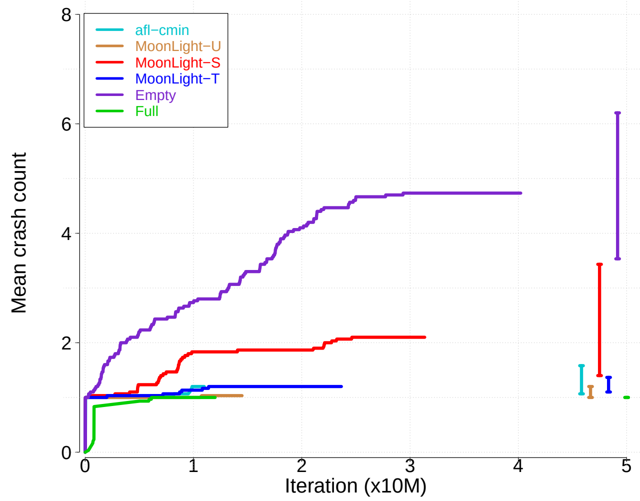
Table 3 displays the distillation results for the corpora, which are consistent with the preliminary results in Section 3.4. MoonLight produces superior corpora when measured against `afl-cmin` whether for weighted or unweighted scenarios. Therefore, the four desirable properties of a fuzzing corpus, as outlined in Section 1, are achieved with MoonLight in a superior way when compared against the current state-of-the-art (`afl-cmin`).

However, what ultimately matters is if this leads to better fuzzing outcomes. Fig. 2 shows the *average fuzzer response* for the seven targets. For each target, there are six response curves shown (the average of 30 campaigns). Each curve corresponds to one of the distillation techniques previously described. Each plot shows the cumulative number of unique crashes found against the test iterations. The intervals displayed on the right-side of each plot show the 95% confidence intervals for the final crash yield. The confidence interval we used is the nonparametric, bias-corrected and accelerated ($BC_a$) bootstrap interval [7]. Uniqueness is determined by stack-hashing with `exploitable` [8].

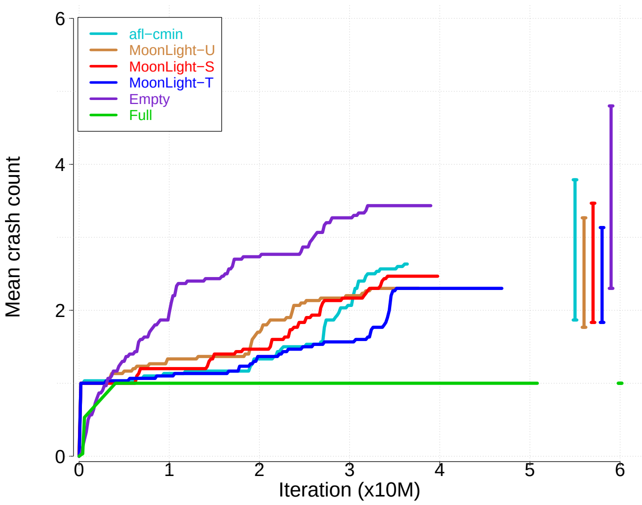
We summarize the main findings from our experiments in Tables 4 and 5. Table 4 shows whether or not a particular corpus distillation technique statistically outperforms `afl-cmin`, and how significant that difference is (using the Vargha-Delaney A measure). Table 4 shows that MoonLight-S statistically outperforms `afl-cmin`—i.e., it either has a strong positive gain or has the same performance. Also clear



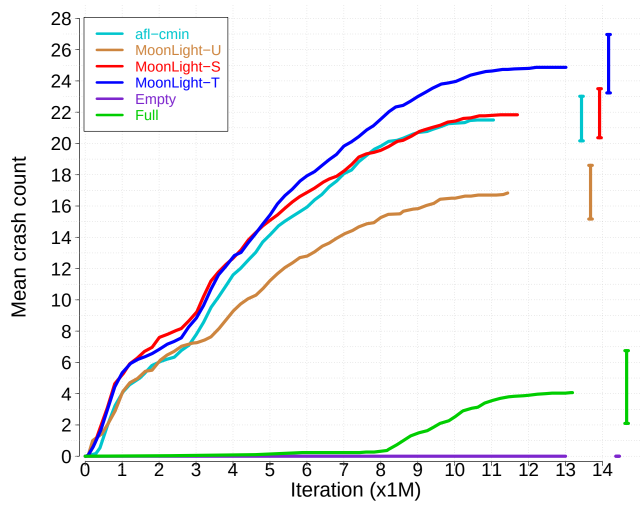

(a) Poppler



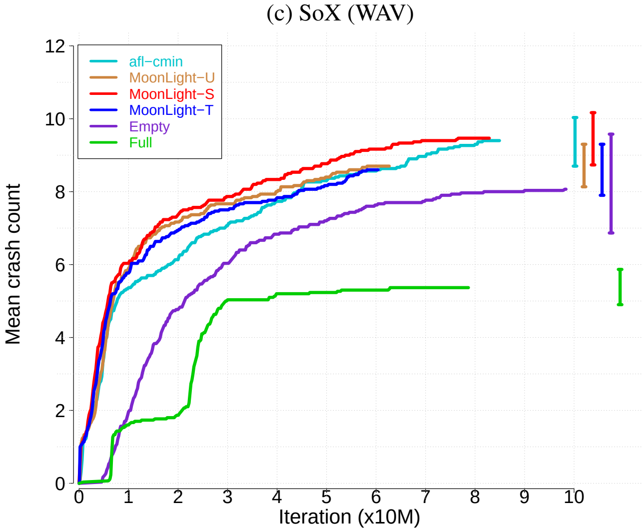

(b) SoX (MP3)



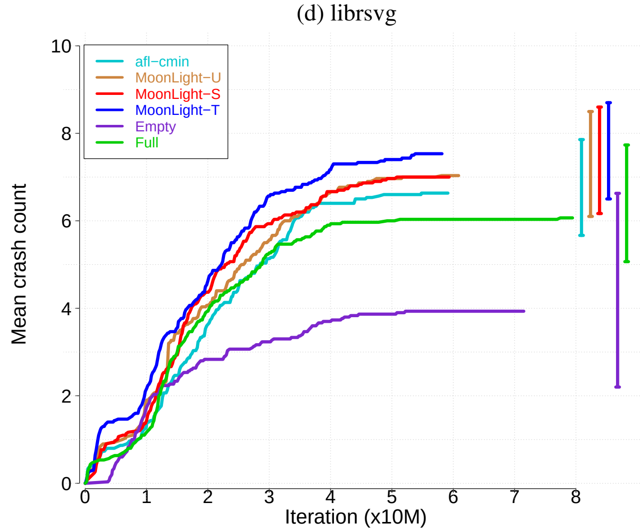

(c) SoX (WAV)




(d) librsvg




(e) libtiff




(f) FreeType

Figure 2: Average unique crashes of thirty 18-hour fuzzing campaigns for each of our 7 targets

Table 3: Comparison of corpora. Each corpus is summarized by its number of files (#); total size, summing the sizes of all included files (MB); and the time taken to run the target over all of the files in turn (s).

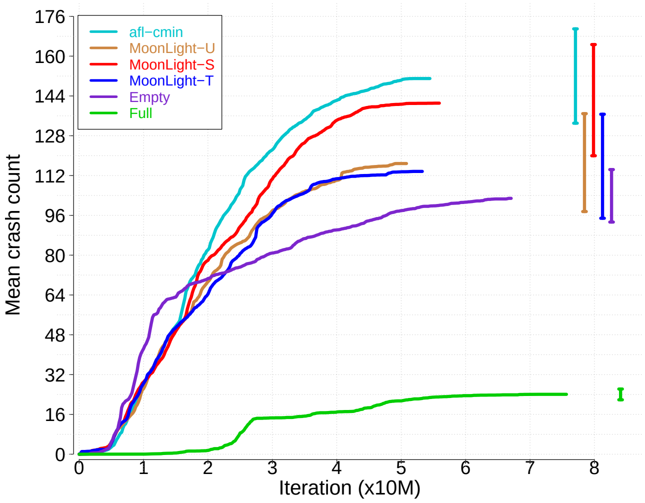
| Program | Collection Corpus | | | `afl-cmin` | | | MoonLight-U | | | MoonLight-S | | | MoonLight-T | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | # | Size (MB) | Time (s) | # | Size (MB) | Time (s) | # | Size (MB) | Time (s) | # | Size (MB) | Time (s) | # | Size (MB) | Time (s) |
| Poppler | 99,984 | 6,086.70 | 7,589.10 | 1,318 | 121.90 | 169.81 | 177 | 22.19 | 28.97 | 209 | 17.32 | 26.85 | 182 | 22.81 | 12.50 |
| SoX (MP3) | 99,691 | 409.44 | 8,028.28 | 137 | 3.75 | 9.70 | 7 | 0.30 | 0.59 | 11 | 0.09 | 0.54 | 9 | 0.47 | 0.13 |
| SoX (WAV) | 74,000 | 249.06 | 548.39 | 68 | 1.65 | 0.52 | 9 | 0.30 | 0.23 | 11 | 0.26 | 0.28 | 9 | 0.27 | 0.23 |
| librsvg | 71,763 | 744.59 | 9,066.77 | 881 | 17.05 | 233.56 | 159 | 3.65 | 49.42 | 183 | 2.58 | 47.94 | 163 | 3.18 | 36.67 |
| libtiff | 99,955 | 44.65 | 1,494.53 | 63 | 0.27 | 0.97 | 21 | 0.09 | 0.31 | 21 | 0.09 | 0.32 | 21 | 0.09 | 0.30 |
| FreeType | 466 | 3.55 | 18.29 | 73 | 8.68 | 2.87 | 23 | 3.00 | 0.93 | 23 | 2.92 | 0.92 | 23 | 3.06 | 0.91 |
| libxml2 | 79,032 | 205.64 | 3,504.85 | 505 | 9.34 | 24.58 | 96 | 1.85 | 5.05 | 120 | 0.96 | 5.98 | 96 | 1.16 | 4.94 |

Table 4: Significance for each corpus distillation technique against `afl-cmin` when comparing unique crashes and AUC. Note: **M** = MoonLight-U, **MS** = MoonLight-S, **MT** = MoonLight-T, and **E** = empty seed. "+" indicates that a distillation is winning against `afl-cmin`, while "-" indicates otherwise. This is followed by the magnitude according to the Vargha-Delaney A measure: "-/+"-Negligible, "--/++"-Small, "---/+++"-Medium, "----/++++"-Large. 0 indicates there is no statistical significance in that comparison.

| Program | M | | MS | | MT | | E | | Full | |
|---|---|---|---|---|---|---|---|---|---|---|
| | crashes | AUC | crashes | AUC | crashes | AUC | crashes | AUC | crashes | AUC |
| Poppler | 0 | -- | +++ | 0 | ++ | 0 | 0 | 0 | ---- | ---- |
| SoX (MP3) | 0 | 0 | 0 | ++++ | 0 | ++++ | ++++ | ++++ | - | ---- |
| SoX (WAV) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ---- | 0 |
| librsvg | ---- | ---- | 0 | 0 | +++ | +++ | ---- | ---- | ---- | ---- |
| libtiff | 0 | ---- | 0 | 0 | -- | --- | -- | 0 | ---- | ---- |
| freetype | 0 | 0 | 0 | 0 | 0 | 0 | ---- | ---- | 0 | 0 |
| libxml2 | -- | --- | 0 | 0 | --- | --- | ---- | ++ | ---- | ---- |
| **Conclusion** | **Bad** | | **Good** | | Depends | | Depends | | **Bad** | |

Table 5: Coefficient of variation ($CV = \sigma/\mu$) for crash yield & AUC. Green cells and yellow cells denote the best performer for each target in terms of crash yield CV and AUC CV respectively. CV$=0$ happens twice with Full, and is an indication of extremely poor yield, so we ignore these cases. Similarly, Empty finds no crashes on librsvg and so $\mu = 0$ and we ignore this also.

| Target | `afl-cmin` | | MoonLight-U | | MoonLight-S | | MoonLight-T | | Empty | | Full | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Crash Yield | AUC | Crash Yield | AUC | Crash Yield | AUC | Crash Yield | AUC | Crash Yield | AUC | Crash Yield | AUC |
| Poppler | 0.26 | 0.43 | 0.18 | 0.43 | 0.29 | 0.62 | 0.22 | 0.56 | 0.61 | 0.67 | 1.89 | 1.81 |
| SoX (MP3) | 0.51 | 0.12 | 0.18 | 0.22 | 1.30 | 1.18 | 0.34 | 0.32 | 0.77 | 0.76 | 0 | 0.28 |
| SoX (WAV) | 0.98 | 0.44 | 0.90 | 0.66 | 0.92 | 0.82 | 0.75 | 0.37 | 1.01 | 1.08 | 0 | 0.08 |
| librsvg | 0.19 | 0.39 | 0.28 | 0.48 | 0.20 | 0.36 | 0.21 | 0.37 | N/A | N/A | 1.60 | 1.58 |
| libtiff | 0.21 | 0.32 | 0.19 | 0.24 | 0.22 | 0.34 | 0.22 | 0.20 | 0.45 | 0.52 | 0.26 | 0.36 |
| FreeType | 0.45 | 0.42 | 0.46 | 0.37 | 0.44 | 0.48 | 0.43 | 0.45 | 1.59 | 1.69 | 0.58 | 0.58 |
| libxml2 | 0.37 | 0.51 | 0.49 | 0.53 | 0.45 | 0.42 | 0.51 | 0.49 | 0.29 | 0.37 | 0.26 | 0.37 |




(g) libxml2

Figure 2 (continued): Average unique crashes of thirty 18 hour fuzzing campaigns

from the table is that using the full corpus results in severe under-performance. Using MoonLight-U is statistically worse than using `afl-cmin` in terms of expected yield performance. Finally, MoonLight-T and the empty corpus are hard to compare: on different targets their performance with respect to `afl-cmin`'s oscillates between good and bad in a statistically significant way. Table 5 shows that using MoonLight-U derived corpus tends to result in fuzzing outcomes with good average yields and low variance. In contrast, when using the empty corpus this is never the case: there is too much variance.

Before drawing general conclusions, we briefly discuss the figures for each target.

In Fig. 2a (Poppler), we see that the full corpus yields twice the number of executions compared to the other approaches. We believe this is due to the fact a very large proportion of seeds in that full corpus for Poppler are extremely fast to execute (without necessarily gaining interesting coverage or crashes). Such seeds clog the fuzzing queue, leading to low productivity. The remainder of the curves perform similarly both in terms of mean yield and yield variance. The empty corpus for Poppler is also quite interesting. Initially, it does not produce any crashes. However, after about one million executions it performs quite well: once AFL has evolved seeds that significantly improve coverage.

In Figs. 2b and 2c (SoX), the number of crashes is relatively small compared to other targets. A highlight is the effectiveness of the empty corpus despite having a large process variance, as observed by the confidence intervals. We also suspect that the relatively smaller iteration counts for these targets mean that the fuzzing process may not have achieved equilibrium within 18 hours.

In Fig. 2d (libsrvg), the empty corpus has a much higher iteration count than indicated on the graph, but all of these iterations had crash counts of zero. MoonLight-U performs particularly worse on this target compared to the other corpus distillation techniques. We believe this may be due to its choice of larger files chosen to minimize the number of files: the average file size in MoonLight-U's corpus is $> 17.5\%$ higher than the other corpus distillation approaches. We speculate that with larger seeds to mutate, many of the mutations are functional duplicates.

The fuzzing results for the FreeType font library have two points of interest. Once again the empty corpus is surprisingly productive, though with a large variance. Unlike the other targets, the full corpus is competitive. Note that full corpus only contains 466 seeds—i.e., it is relatively small. This suggests that distillation is only worthwhile when there are many seeds in the corpus.

Fig. 2g (libxml2) illustrates an interesting contrast between crash yield and the AUC measure (as reported in Table 4). While the empty corpus ends the campaign with fewer crashes found (statistically significantly), its early successes and greater iteration count actually mean that it is better, again statistically significantly, in terms of AUC.

Finally, it is worth considering the results ranked by variance rather than yield. Approaches which have smaller variance are valuable because their performance is less risky. For example, the empty corpus can give astonishingly high yields but is also very risky because the variance is so large. Table 5 shows the coefficient of variation for crash yield and AUC for our experiments. The highlighted coefficient of variation is the lowest for that target, indicating the distillation approach with the most consistent behavior.

## 4.3 Findings

We now summarize our findings.

**MoonLight-S.** Overall this technique statistically outperforms `afl-cmin` and other approaches in terms of expected yield performance. The benefit over `afl-cmin`, however, was only observed against two targets otherwise the evidence was insignificant. This result is consistent with our theory and the polynomial reduction in mutation search space described earlier.

**Full corpus.** The full corpus technique is recommended only when the total number of seeds is small—i.e., in the order of hundreds or less. When there are thousands of seeds in the collection corpus it is imperative that some form of distillation is applied.

**Empty corpus.** The empty corpus is a surprising performer on average, however, individual trials may differ wildly from the mean. That said, perhaps it makes sense to always add the empty seed to any fuzzing corpus and rely on the fuzzer's

Table 6: Triaging SoX: Number of trials that found given bugs. Note: **MS** = MoonLight-S.

| Bug Id | CWE | afl-cmin | MS | CVE |
|---|---|---|---|---|
| A | 130 | 30 | 30 | – |
| B | 680 | 1 | 3 | 2019-8355 |
| C | 122 | 2 | 4 | – |
| D | 690 | 0 | 3 | 2019-8357 |
| E | 680 | 0 | 3 | 2019-8354 |
| F | 121 | 0 | 3 | 2019-8356 |
| G | 476 | 3 | 3 | 2017-18189 |

own reinforcement learning to decide if the empty seed is valuable or not.

**MoonLight-T.** The performance of this distillation technique was found to be highly target dependent. For certain targets it is a very strong performer, but an equally strong under-performer for other targets. It would make sense to use this strategy if you plan a long campaign and have time to do some trials against the target before hand. Otherwise choose MoonLight-S.

**MoonLight-U.** This strategy performed slightly worse than `afl-cmin` with respect to yield but demonstrated strong performance with respect to variance. MoonLight-U is the most consistent corpus for most of the targets. In most cases (SoX on WAV files, FreeType, and libxml2) where it did not achieve the lowest coefficient of variation, its score is comparable to the other distillation techniques. A conservative fuzzing campaign where variance minimization is more important than expected yield returns would benefit by using this distillation technique.

## 4.4 Triage

We have already presented sufficient data to answer the research questions raised in Section 1. Ultimately, however, the ideal performance measure for fuzzers is the number of distinct bugs that fuzzers find [15], and so we debug the crashes from one target (SoX on MP3 files). We find seven bugs, some of them security-interesting. Of these seven, one was previously reported (CVE-2017-18189). Time and space constraints prevented us from analyzing the remaining targets.

Since we have established that MoonLight-S outperforms `afl-cmin`, we compare just those two distillation techniques. Collectively for this target, seven unique bugs were found which we label A–G:

**A** Overlapping source and destination addresses;

**B** Integer overflow causes improper allocation on heap buffer;

**C** Heap buffer overflow in a 3rd party library used by SoX (libmad), due to malformed arguments passed;

**D** Integer overflow causes failed memory allocation;

**E** Integer overflow causes failed memory allocation;

**F** Stack buffer bounds violation; and

**G** Null pointer dereference (previously reported CVE-2017-18189).

The counts for how many times each bug was found within the 30 campaigns of our experiments appear in Table 6. In particular, note that the MoonLight-distilled corpus finds three bugs (D–F) (three times each) that `afl-cmin` does not. We have reported and received CVEs for four of the vulnerabilities.

## 5 Conclusions

Our premise is that the choice of fuzzing corpus is a critical decision made before a fuzzing campaign begins. This work provides ample confirmation that this is indeed the case.

We have performed extensive experiments (20.7 CPU-years worth) to produce findings that have a strong statistical support for our claims. On the basis of theoretical reasoning about mutation-based fuzzing, we developed a near-optimal algorithm for the weighted minimum set cover problem. We further predicted that distillation using file size weighting would significantly reduce the mutation search space and result in more effective fuzzing. This was shown to be the case. Indeed, we outperform the current state-of-the-art tool, `afl-cmin`, when we distill by file sizes. Our tool, MoonLight, has been open-sourced along with some of our collection corpus trace data [6] and the MoonBeam coverage tracing tool.

We add to the knowledge of how to perform effective fuzzing in practice:

- Maximizing fuzzing yield is achieved by using MoonLight-S (weighted by file sizes).
- Minimum variance fuzzing is achieved through the use of MoonLight-U (unweighted distillations).
- Avoid fuzzing with a large collection corpus—i.e., on the order of a thousand files or more. Conversely, if the collection corpus is small, then distillation is not helpful.

We also triaged the crashes from one of our targets, finding seven bugs, five of which were security-interesting, and four that are new. Moreover, MoonLight-S's corpus led to the discovery of three security-interesting bugs that the `afl-cmin` corpus did not find. We have reported all four new bugs and received CVEs for them.

**Future work.** Some of our experiments raise new questions in response to observed unexpected behaviors. For example, the performance of the empty corpus and the corpus distilled by execution times show unexpected volatility. Depending on the target, these approaches can show outstanding performance or the opposite. The reasons are unclear and require further investigation. It would be valuable to practitioners to know *a priori* to campaigning if these approaches would be effective.

[6] Available at https://bit.ly/2JJfffY

## Acknowledgements

This work was supported in part by a grant from the Next Generation Technologies Fund, managed by the Defence Science and Technology Group, through Data61.

## References


[1] Humberto Abdelnur, Radu State, Obes Jorge Lucangeli, and Olivier Festor. Spectral fuzzing: Evaluation & feedback. Research Report RR-7193, INRIA, February 2010.

[2] Laurent Andrey, Humberto Abdelnur, Jorge Lucangeli Obes, Olivier Festor, and Radu State. Deliverable D2.1 closed loop fuzzing algorithms, August 2010. ANR-08-VERS-017 (Vampire) project.

[3] R.E. Bellman. *Dynamic Programming*. Dover Books on Computer Science. Dover Publications, 2003.

[4] Oliver Chang, Abhishek Arya, Kostya Serebryany, and Josh Armour. Oss-fuzz: Five months later, and rewarding projects. https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017.

[5] Vasek Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[6] Deja Vu Security. Peachminset. http://community.peachfuzzer.com/v3/minset.html.

[7] Bradley Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171–185, 1987.

[8] Jonathan Foote. Exploitable. https://github.com/jfoote/exploitable, 2011.

[9] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy*, pages 679–696, San Francisco, California, May 2018.

[10] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *ACM/IEEE International Conference on Software Engineering*, pages 72–82, Hyderabad, India, May/Jun. 2014.

[11] Gustavo Grieco, Martín Ceresa, Agustín Mista, and Pablo Buiras. QuickFuzz testing for fun and profit. *Journal of Systems and Software*, 134:340–354, 2017.

[12] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification, and Validation*, pages 243–252, Cleveland, Ohio, Mar./Apr. 2014.

[13] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Symposium on Network and Distributed System Security*, 2019.

[14] R. M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68, January 1975.

[15] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, Toronto, Canada, October 2018.

[16] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 560–564, Montréal, Canada, March 2015.

[17] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *ESEC/SIGSOFT Joint Meeting on Foundations of Software Engineering*, pages 627–637, Paderborn, Germany, September 2017.

[18] Chenyang Lv, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, Pan Zhou, and Jing Chen. SmartSeed: Smart seed generation for efficient fuzzing. *CoRR*, abs/1807.02606, 2018.

[19] Charlie Miller. Fuzz by number: More data about fuzzing than you ever wanted to know. In *CanSecWest*, March 2008. https://cansecwest.com/csw08/csw08-miller.pdf.

[20] Mozilla. Dharma: A generation-based, context-free grammar fuzzer. https://blog.mozilla.org/security/2015/06/29/dharma/, 2015.

[21] Ben Nagy. Prospecting for rootite. In *Ruxcon*, 2010. https://fuzzinginfo.files.wordpress.com/2012/05/ben-nagy-prospecting-for-rootite-2010.pdf.

[22] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *USENIX Security Symposium*, pages 729–743, Baltimore, Maryland, August 2018.

[23] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, pages 861–875, San Diego, California, August 2014.

[24] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys. In *Annual Meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[25] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, Boston, Massachusetts, June 2012. USENIX.

[26] Kosta Serebryany. Continuous fuzzing with libFuzzer and AddressSanitizer. In *IEEE Cybersecurity Development*, SecDev, page 157, Boston, Massachusetts, November 2016.

[27] Robert Swiecki. honggfuzz. http://honggfuzz.com/, 2016.

[28] András Vargha and Harold D Delaney. A critique and improvement of the CL common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[29] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy*, pages 579–594, San Jose, California, May 2017.

[30] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, Dallas, Texas, October 2017.

[31] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, pages 745–761, Baltimore, Maryland, August 2018.

[32] Michał Zalewski. American fuzzy lop (afl). http://lcamtuf.coredump.cx/afl/, 2015.

[33] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

## A Appendices

### A.1 Moonlight Theory

**Problem Definition.** Recall that our target has a finite set of edges between its constituent basic blocks, and any given seed will cause a subset of these edges to be traversed when the target is executed on that seed. This seed's set of edges is its *coverage*, and the coverage of a corpus is the union of all that set's coverages.

Our algorithm applies dynamic programming to take a large coverage matrix and recursively transform it through row and column eliminations into successively smaller matrices whilst accumulating a minimum cover set $\mathbb{C}$.

What follows is an elaboration of the key theoretical ideas that support our algorithm along with relevant supporting proofs. Firstly, we extend the notation defined in Section 3 with the following:

- $a_{i,j} = \mathbb{I}[\textit{seed } i \text{ uses } \textit{edge } j]$ where $\mathbb{I}$ is the *indicator* function; and
- $x_i = \mathbb{I}[s_i \text{in the computed solution}]$ be the decision vector for seeds that describes a solution.

### A.2 Matrix Operations

Before presenting the dynamic programming formulation and solution for this problem we need to define our *operational primitives* which allow us to transform the large original problem into smaller over-lapping sub-problems.

We introduce our basic operations as follows:

- $\rho$ denotes a subset of rows $S$ in $A$;
- $\kappa$ denotes a subset of columns $B$ in $A$;
- $A' = A\backslash_{row}\rho$ denotes the matrix $A$ with the rows $\rho$ removed. We term this *row elimination*;
- $A' = A\backslash_{col}\kappa$ denotes the matrix $A$ with the columns $\kappa$ removed. We term this *column elimination*;
- Function $\Delta(A,\rho,\kappa) = A\backslash_{row}\rho\backslash_{col}\kappa$, which is the matrix $A$ with rows $\rho$ and columns $\kappa$ eliminated. We term this a *matrix reduction*; and
- $minset(A)$ is the *size* of the minimum cover set of matrix $A$.

We proceed with the working example in Fig. 1a to describe our algorithmic operations.

**Singularities.** We define the term *singularity* to describe the situation where a column or row sum is *zero*. Additionally, we define the following:

- Let $\kappa_\theta = \theta(A)$ be the set of all *singular columns* of matrix $A$; and
- Let $A' = A\backslash_{col}\kappa_\theta$ be the matrix $A$ with the column singularities removed.

LEMMA: The matrix $A' = A\backslash_{col}\kappa_\theta$ has the same minimum cover set as $A$. That is, $minset(A) = minset(A')$.

PROOF: Let the set of columns of $A'$ be $\kappa'$ and the set of columns of $A$ be $\kappa$. Therefore $\kappa = \kappa' \bigcup \kappa_\theta$. However no row exists that covers any column in $\kappa_\theta$ otherwise the column would not be singular. So any minimum cover set of $A$ does not cover columns $\kappa_\theta$ and so $\kappa_\theta$ is superfluous to the computation. Therefore any minimum cover set of $A$ must also be a minimum cover set of $A'$.

Consider Fig. 1b from the working example. Column $e_6$ highlighted in red is a singular column. We can eliminate this column to produce a smaller matrix $A'$ whose minimum cover set is the same as the original matrix $A$.

**Exotic Rows.** An *exotic column* $e^*$ in $A$ has a column sum of *one*. An *exotic row* is the unique row $s^*$ in $A$ that covers $e^*$. For example in Fig. 1c the row $s_1$ is exotic because it is the only row that covers column $e_3$.

LEMMA: An exotic row is a member of any minimum cover set.

PROOF: Almost by definition. Any minimum cover set of $A$ must include rows that cover all the non-singular columns. An exotic row *uniquely* covers at least one exotic column in $A$ which forces its selection as a member of a minimum cover set.

- Let $S_\chi$ be the set of all exotic rows in $A$; and
- Let $A' = A\backslash_{row}S_\chi$.

LEMMA: $minset(A) = minset(A') + \|S_\chi\|$

PROOF: Let $s_i$ be an exotic row in $A$. Therefore $s_i \in S_\chi$ by definition. For convenience let $x = minset(A)$ and $y = minset(A')$. Therefore $x = y+1$ since $s_i$ must be in the optimal cover set and $A$ and $A'$ differ by only one row: the exotic row $s_i$. By induction for all rows in $S_\chi$ the lemma follows.

**Dominant Rows.** Row dominance captures the intuitive idea that some rows in $A$ may be a subset of a single row. We will need to consider row weights in this operation. We make this notion more precise as follows:

- Let $s_\alpha$ be a row that covers the set of columns $\alpha \subseteq E$;
- Let $s_\beta$ be a row that covers the set of columns $\beta \subseteq \alpha$;
- We say that $s_\alpha$ *dominates* $s_\beta$;
- We say that $s_\beta$ *submits* to $s_\alpha$;
- If $s^*_\alpha$ has no dominators then we call it a *primal dominator*;
- Let $w^*_\alpha$ be the *weight* of $s^*_\alpha$;
- Let $S_\lambda$ be the set of *all* rows that are dominated by $s^*_\alpha$ *and whose individual row weights are less than or equal to* $s^*_\alpha$. We say that $S_\lambda$ is a *submissive set* of rows; and
- Let $A' = A\backslash_{row}S_\lambda$ be the matrix with the submissive rows from $A$ removed.

Continuing with the (unweighted) working example in **??** we identify row $s_5$ as a dominant row—shown in red—since it dominates rows $S_\lambda = \{s_2, s_3, s_4\}$—shown in yellow.

In the case of *unweighted* distillations, row dominance and submission strictly captures the idea that submissive rows are a proper subset cover of a dominant row. In the *weighted* case only rows that are a proper subset of a dominator *and have a larger weight* then the dominator can be removed. Submissive rows with a smaller weight could in principle be combined with another row whose combined weight is less than the dominant row and therefore should not be removed.

LEMMA: $minset(A') = minset(A)$

PROOF: For convenience let $x = minset(A)$ and $y = minset(A')$. Since $A$ contains the primal dominator row $s^*_\alpha$ none of the submissive rows in $S_\lambda$ can be in the weighted minimum cover set. If they were we could remove them to make a smaller weighted cover set giving a contradiction. Therefore $x$ does not count the submissive rows. Additionally, $A'$ does not contain the submissive rows since they were eliminated. Therefore $y$ could not include a count of these rows. All else is equal so the lemma follows.

Eliminating a submissive row set allows us to transform a larger matrix into a smaller one. It is a very effective dimensional reduction technique in practice. The fact that *weighted* row eliminations do not remove as many rows as the *unweighted* case will explain why weighted minimum cover sets are usually larger (i.e., have more seeds) than unweighted ones.

**Dominant Columns.** Column dominance is a similar idea to row dominance. In this case some columns in $A$ may be a subset of a single column.

- Let $e_\alpha$ be a column that covers the set of rows $\alpha \subseteq S$;
- Let $e_\beta$ be a column that covers the set of rows $\beta \subseteq \alpha$;
- We say that $e_\alpha$ *dominates* $e_\beta$;
- We say that $e_\beta$ *submits* to $e_\alpha$;
- Let $E_\mu$ be the set of all columns dominated by column $e_\alpha$; and
- Let $A' = A \backslash_{col} e_\alpha$.

Continuing with the working example in Fig. 1e we identify column $e_5$ as a dominant column since it dominates columns $E_\mu = \{e_1, e_3, e_4\}$. Note that $e_2$ is not dominated by $e_5$ because of $s_2$.

LEMMA: $minset(A') = minset(A)$

PROOF: The minimum cover set $S^*(A)$ must contain seeds which cover all of $S_\alpha$. However any seed that covers a column in $S_\alpha$ must also cover $e_\alpha$ since $e_\alpha$ is a dominant column. Therefore any minimum cover set of $A' = A \backslash_{col} e_\alpha$ is also going to be a minimum cover set of $A$ since the dominant column is redundant in this case.

Eliminating dominant columns when they exist is an effective dimensional reduction technique that comes with no penalty.

**Contained Columns.** When choosing row $s_i$ to add to our solution we can eliminate all the columns that $s_i$ covers.

- Let $S_i$ be all the columns in $A$ that $s_i$ covers;
- Let $A' = A \backslash_{row} s_i$; and
- Let $A'' = A' \backslash_{col} S_i$.

LEMMA: $minset(A) - minset(A'') \leq 1$

PROOF: For convenience let $x = minset(A)$ and $y = minset(A')$. Now there are two cases: if the selected row elimination is an optimal elimination then $x = y$. If the selected row that was eliminated is *not* a member of a minimum cover set then $x = y + 1$. When we remove the columns $S_i$ we know that $s_i$ covers them. Therefore these columns are no longer needed in calculating the solution. Therefore $minset(A'') = minset(A') = y$. Therefore $x - y \leq 1$ and the lemma follows.

We call this operation *contained column eliminations*. Again it is a very powerful technique for reducing the dimensionality of the problem.

**Heuristic Row Reduction.** So far the row and column eliminations that we have described have been optimal in the sense that we have been able to guarantee that an optimal solution for a smaller transformed matrix can be used to construct an optimal solution for the larger matrix.

There are times when you exhaust all optimal transformations and you need to make a *heuristic choice*.

Intuitively, a good choice is to select the row with the largest *rowsum* since it will trigger a large *contained columns elimination*. In the weighted case we choose the row with with the largest *rowsum*/*rowweight*.

The expectation is that optimal transformations can be resumed on the smaller matrix.

## A.3 MoonLight Algorithm

We now present a more formal description of the MoonLight algorithm.

**Bellman Equation.** The Bellman Equation is the necessary condition for optimality when using dynamic programming. We define the *state* of the problem to be specified by the coverage matrix $A$. The *cost* of the state is defined to be the cost of any matrix operation performed on $A$ denoted by $d_\star(A, A')$ plus the cost of the subsequent state $A'$. Clearly we want to choose the optimum transformation for state $A$. Therefore we define the Bellman equation for the problem as:

$$Cost(A) = \min_{i \in \{Operators\}} \begin{cases} d_i(A, A') + Cost(A') & A \neq Z \\ 0 & A = Z \end{cases} \quad (2)$$

where the indexation consists of the five unary operations previously specified. We also make clear that any transformation from $A \to A'$ implicitly uses the matrix reduction operation $\Delta(\cdot)$ imposed on any associated contained columns $\kappa = \Pi(\cdot)$. This will be discussed momentarily.

What are the costs $d_\star$ for each operation? The objective is to find an *optimum* set cover for the problem. Optimal reductions occur when $minset(A) = minset(A')$. We specify these to have a cost of zero if and only if the associated row or columns sets are not empty (that is, $A \neq A'$), else they have an infinite cost. Similarly, exotic row operations are also defined to have a zero cost even though the before and after minimum cover sets are different. For example, removing column singularities and selecting exotics row are all optimal operations and incur no cost providing they reduce the size of the matrix $A'$.

In contrast heuristic row reductions can not be guaranteed to be optimal reductions. Therefore we define the cost of this operation to be the *number of rows* selected—that is $|\varepsilon(A)|$. This cost represents an upper bound measure of divergence from the size of the optimal choice at state $A$. That is $|minset(A')| - |minset(A)| \leq |\varepsilon(A)|$. This is an inductive argument: every time you choose a row heuristically it may not be an optimal choice. Therefore counting the number of heuristic choices measures the size of the deviation from an optimal solution.

## A.4 The "Empty Seed" Corpus

As discussed in Section 4.1, we use a small, hand-constructed input when fuzzing the "empty seed". For most targets, the empty seed contains a single line-break character ("\n"). However, for more structured inputs—e.g., SVG, WAV, and PDF—we use a slightly more complex empty seed to assist AFL in overcoming magic byte checks. For example, the "empty" SVG and PDF seeds are shown in Fig. 3.

```
<svg></svg>
```

(a) The empty SVG seed

```
%PDF-1.7
1 0 obj
<< /Type  /Catalog
>>
endobj
trailer
<<
/Root 1 0 R
>>
%%EOF
```

(b) The empty PDF seed

Figure 3: "Empty" seed examples