Optimizing Seed Selection for Fuzzing

Alexandre Rebert[‡,$] alex@forallsecure.com
Sang Kil Cha[‡] sangkilc@cmu.edu
Thanassis Avgerinos[‡] thanassis@cmu.edu
Jonathan Foote[†] jmfoote@cert.org
David Warren[†] dwarren@cert.org
Gustavo Grieco[§] gg@cifasis-conicet.gov.ar
David Brumley[‡] dbrumey@cmu.edu

[‡] Carnegie Mellon University [$] ForAllSecure, Inc. [§] CIFASIS-CONICET
[†] Software Engineering Institute CERT

Abstract

Randomly mutating well-formed program inputs or simply *fuzzing*, is a highly effective and widely used strategy to find bugs in software. Other than showing fuzzers find bugs, there has been little systematic effort in understanding the science of how to fuzz properly. In this paper, we focus on how to mathematically formulate and reason about one critical aspect in fuzzing: how best to pick seed files to maximize the total number of bugs found during a fuzz campaign. We design and evaluate six different algorithms using over 650 CPU days on Amazon Elastic Compute Cloud (EC2) to provide ground truth data. Overall, we find 240 bugs in 8 applications and show that the choice of algorithm can greatly increase the number of bugs found. We also show that current seed selection strategies as found in Peach may fare no better than picking seeds at random. We make our data set and code publicly available.

1 Introduction

Software bugs are expensive. A single software flaw is enough to take down spacecrafts [2], make nuclear centrifuges spin out of control [17], or recall 100,000s of faulty cars resulting in billions of dollars in damages [5]. In 2012, the software security market was estimated at $19.2 billion [12], and recent forecasts predict a steady increase in the future despite a sequestering economy [19]. The need for finding and fixing bugs in software before they are exploited by attackers has led to the development of sophisticated automatic software testing tools.

Fuzzing is a popular and effective choice for finding bugs in applications. For example, fuzzing is used as part of the overall quality checking process employed by Adobe [28], Microsoft [14], and Google [27], as well as by security companies and consultants to find bugs and vulnerabilities in COTS systems.

One reason fuzzing is attractive is because it is relatively straightforward and fast to get working. Given a target application P, and a set of seed input files S, the programmer needs to:

Step 1. Discover the command line arguments to P so that it reads from a file. Popular examples include `-f`, `-file`, and using `stdin`. This step can be manual, or automated in many cases using simple heuristics such as trying likely argument combinations.

Step 2. Determine the relevant file types for an application automatically. For example, we are unlikely to find many bugs fuzzing a PDF viewer with a GIF image. Currently this step is performed manually, and like the above step the manual process does not scale to large program bases.

Step 3. Determine a subset of seeds $S' \subseteq S$ to fuzz the program. For example, an analyst may consider the possible set of seeds S as every PDF available from a search engine. Clearly fuzzing on each seed is computationally prohibitive, thus a *seed selection strategy* is necessary. Two typical choices are choosing the set of seed files ad-hoc, e.g., those immediately handy, and by finding the minimal set of seeds necessary to achieve code coverage.

Step 4. Fuzz the program and reap risk-reducing, or profitable, bugs.

Throughout this paper we assume maximizing the number of unique bugs found is the main goal. We make no specific assumptions about the type of fuzzer, e.g., we do not assume nor care whether black-box, white-box, mutational, or any other type of fuzzing is used. For our

experiments, we use BFF, a typical fuzzer used in practice, though the general approach should apply to any fuzzer using seeds. Our techniques also make no specific assumptions about the fuzz scheduling algorithm, thus are agnostic to the overall fuzzing infrastructure. To evaluate seed selection strategies, we use popular scheduling algorithms such as round-robin, as well as the best possible (optimal) scheduling.

We motivate our research with the problem setting of creating a hypothetical fuzzing testbed for our system, called COVERSET. COVERSET periodically monitors the internet, downloads programs, and fuzzes them. The goal of COVERSET is to maximize the number of bugs found within a limited time period or budget. Since budgets are forever constrained, we wish to make intelligent design decisions that employ the optimal algorithms wherever possible. How shall we go about building such a system?

Realizing such an intelligent fuzzing system highlights several deep questions:

Q1. Given millions, billions, or even trillions of PDF files, which should you use when fuzzing a PDF viewer? More generally, what algorithms produce the best result for seed selection of $S' \subseteq S$ in step 3?

Q2. How do you measure the quality of a seed selection technique *independently* of the fuzzing scheduling algorithm? For example, if we ran algorithm A on seed set S_1 and S_2, and S_1 maximized bugs, we would still be left with the possibility that with a more intelligent scheduling algorithm A' would do better with S_2 rather than S_1. Can we develop a theory to justify when one seed set is better than another with the best possible fuzzing strategy, instead of specific examples?

Q3. Can we converge on a "good" seed set for fuzzing campaigns on programs for a particular file type? Specifically, if S' performs well on program P_1, how does it work on other similar applications $P_2, P_3, \ldots$? If there is one seed set that works well across all programs, then we would only need to precompute it once and forever use it to fuzz any application. Such a strategy would save immense time and effort in practice. If not, we will need to recompute the best seed set for each new program.

Our main contribution are techniques for answering the above questions. To the best of our knowledge, many of the above problems have not been formalized or studied systematically. In particular:

- We formalize, implement, and test a number of existing and novel algorithms for seed selection.
- We formalize the notion of ex post facto optimality seed selection and give the first strategy that provides an optimal algorithm even if the bugs found by different seeds are correlated.
- We develop evidence-driven techniques for identifying the quality of a seed selection strategy with respect to an optimal solution.
- We perform extensive fuzzing experiments using over 650 CPU days on Amazon EC2 to get ground truth on representative applications. Overall, we find **240** unique bugs in 8 widely-used applications, all of which are on the attack surface (they are often used to process untrusted input, e.g., images, network files, etc.), most of which are security-critical.

While our techniques are general and can be used on any data set (and are the main contribution of this work), our particular result numbers (as any in this line of research) are data dependent. In particular, our initial set of seed files, programs under test, and time spent testing are all important factors. We have addressed these issues in several ways. First, we have picked several applications in each file type category that are typical of fuzzing campaigns. This mitigates incorrect conclusions from a non-representative data set or a particularly bad program. Second, we have performed experiments with reasonably long running times (12 hour campaigns per file), accumulating over 650 CPU days of Amazon EC2 time. Third, we are making our data set and code available, so that: 1) others need not spend time and money on fuzzing to replicate our data set, 2) others can further analyze the statistics to dig out additional meaning (e.g., perform their own hypothesis testing), and 3) we help lower the barrier for further improvements to the science of vulnerability testing and fuzzing. For details, please visit: http://security.ece.cmu.edu/coverset/.

2 Q1: Seed Selection

How shall we select seed files to use for the fuzzer? For concreteness, we downloaded a set of seed files S consisting of $4,912,142$ distinct files and 274 file types from Bing. The overall database of seed files is approximately 6TB. Fuzzing each program for a sufficient amount of time to be effective across all seed files is computationally expensive. Further, sets of seed files are often duplicative in the behavior elicited during fuzzing, e.g., s_1 may produce the same bugs as s_2, thus fuzzing both s_1 and s_2 is wasteful. Which subset of seed files $S' \subseteq S$ shall we use for fuzzing?

Several papers [1, 11], presentations from well-respected computer security professionals [8, 22, 26], as well as tools such as Peach [9], suggest using executable code coverage as a seed selection strategy. The intuition is that many seed files likely execute the same code blocks,

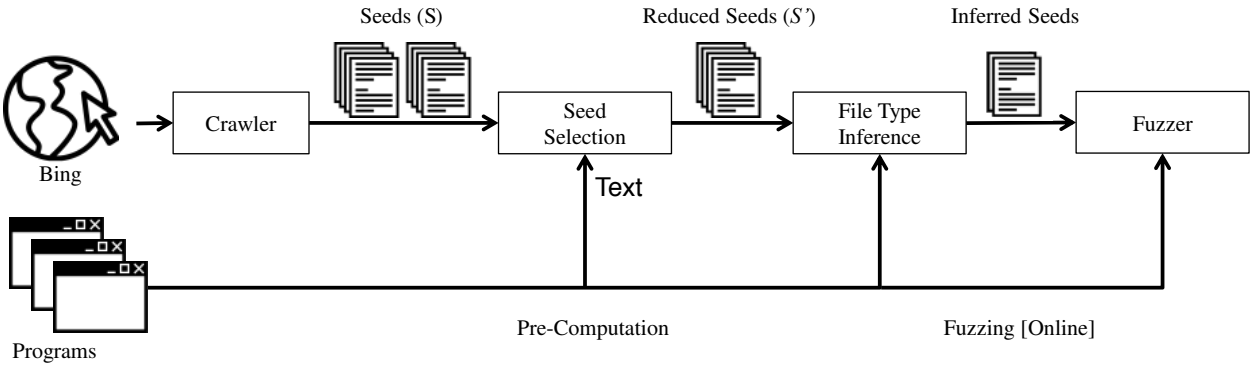

Figure 1: The COVERSET pipeline.

and such seeds are likely to produce the same bugs. For example, Miller reports a 1% increase in code coverage increases the percentage of bugs found by .92% [22]. This intuition can be formalized as an instance of the set cover problem [1, 11]. Does set cover work? Is the minimal set cover better than other set covers? Should we weight the set cover, e.g., by how long it takes to fuzz a particular seed? Previous work has shown a correlation between coverage and bugs found, but has not performed *comparative* studies among a number of approaches, nor studied how to measure optimality (§ 3).

Recall that in the *set cover problem* (SCP) [6] we are given a set X and a finite list of subsets $\mathbb{F} = \{S_1, S_2, \ldots, S_n\}$ such that every element of X belongs to at least one subset of $\mathbb{F}$:

$$X = \bigcup_{S \in \mathbb{F}} S$$

We say that a set $\mathbb{C} \subseteq \mathbb{F}$ is a set cover of X when:

$$X = \bigcup_{S \in \mathbb{C}} S$$

The seed selection strategy is formalized as:

Step 1. The user computes the coverage for each of the n individual seed files. The output is the set of code blocks [1] executed per seed. For example, suppose a user is given $n = 6$ seeds such that each seed executes the following code blocks:

$$S_1 = \{1,2,3,4,5,6\} \quad S_2 = \{5,6,8,9\}$$
$$S_3 = \{1,4,7,10\} \quad S_4 = \{2,5,7,8,11\}$$
$$S_5 = \{3,6,9,12\} \quad S_6 = \{10,11\}$$

Step 2. The user computes the cummulative coverage $X = \bigcup S_i$, e.g., $X = \{1,2,\ldots,12\}$ for the above.

[1] We assume code blocks, though any granularity of unit such as instruction, function, etc. also work.

Step 3. The user computes a set cover to output a subset $\mathbb{C}$ of seeds to use in a subsequent fuzzing campaign. For example, $\mathbb{C}_1 = \{S_1, S_4, S_3, S_5\}$ is one set cover, as is $\mathbb{C}_2 = \{S_3, S_4, S_5\}$, with $\mathbb{C}_2$ being optimal in the unweighted case.

The goal of the *minimal set cover problem* (MSCP) is to minimize the number of subsets in the set cover $\mathbb{C} \subseteq \mathbb{F}$. We call such a set $\mathbb{C}$ a *minset*. Note that a minset need not be unique, i.e., there may be many possible subsets of equal minimal cardinality. Each minset represents the fewest seed files needed to elicit the maximal set of instructions with respect to S, thus represents the maximum data seed reduction size.

In addition to coverage, we may also consider other attributes, such as speed of execution, file size, etc. A generalization of the set cover is to include a weight $w(S)$ for each $S \in \mathbb{F}$. The total cost of a set cover $\mathbb{C}$ is:

$$Cost\,(\mathbb{C}) = \sum_{S \in \mathbb{C}} w(S)$$

The goal of the *weighted* minimal set cover problem (WMSCP) is to find the minimal cost cover set, i.e., $\underset{\mathbb{C}}{argmin}\ Cost\,(\mathbb{C})$.

Both the MSCP and WMSCP can be augmented to take an optional argument k (forming k-SCP and k-WSCP respectively) specifying the maximum size of the returned solution. For example, if $k = 2$ then the number of subsets is restricted to at most 2 ($|\mathbb{C}| \leq 2$), and the goal is to *maximize* the number of covered elements. Note the returned set may not be a complete set cover.

Both MSCP and WMSCP are well-known NP-hard problems. Recall that a common approach to dealing with NP-hard problems in practice is to use an approximation algorithm. An approximation algorithm is a polynomial-time algorithm for approximating an optimal solution. Such an algorithm has an *approximation ratio* $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of

an optimal solution. The minimal set cover and weighted set cover problems both have a *greedy* polynomial-time $\ln|X|+1$-approximation algorithm [4, 16], which is a threshold below which set cover cannot be approximated efficiently assuming NP does not have slightly super-polynomial time algorithms, i.e., the greedy algorithm is essentially the best algorithm possible in terms of the approximation ratio it guarantees [10]. Since $\ln|X|$ grows relatively slowly, we expect the greedy strategy to be relatively close to optimal.

The optimal greedy polynomial-time approximation algorithm[2] for WSCP is:

GREEDY-WEIGHTED-SET-COVER$(X,\mathbb{F})$

```
1  U = X
2  ℂ = ∅
3  while U ≠ ∅
4      S = argmax_{S∈𝔽} |S∩U| / w(S)
5      ℂ = ℂ ∪ S
6      U = U \ S
7  return ℂ
```

Note that the unweighted minset can be solved using the same algorithm by setting $\forall S : w(S) = 1$.

2.1 Seed Selection Algorithms

In this section we consider: the set cover algorithm from Peach [9], a minimal set cover [1], a minimal set cover weighted by execution time, a minimal set cover weighted by size, and a hotset algorithm. The first two algorithms have previously been proposed in literature; the remaining are additional design points we propose and evaluate here. We put these algorithms to the test in our evaluation section to determine the one that yields the best results (see § 6).

All algorithms take the same set of parameters: given $|\mathbb{F}|$ seed files, the goal is to calculate a data reduction to k files where $k \ll |\mathbb{F}|$. We assume we are given t seconds to perform the data reduction, after which the selected k files will be used in a fuzzing campaign (typically of much greater length than t). We break ties between two seed files by randomly choosing one.

PEACH SET. Peach 3.1.53 [9] has a class called MinSet that calculates a cover set $\mathbb{C}$ as follows: [3]

PEACH-MINSET$(P,\mathbb{F})$

```
1  ℂ = ∅
2  i = 1
3  for S in 𝔽
4      cov[i] = MeasureCoverage( S )
5      i = i + 1
6  sort( cov ) // sort seeds by coverage
7  for i = 1 to |𝔽|
8      if cov[i] \ ℂ ≠ ∅
9          ℂ = ℂ ∪ cov[i]
10 return ℂ
```

Despite having the name MinSet, the above routine does not calculate the minimal set cover nor a proven competitive approximation thereof.

RANDOM SET. Pick k seeds at random. This approach serves as a baseline for other algorithms to beat. Since the algorithm is randomized, RANDOM SET can have high variance in terms of seed quality and performance. To measure the effectiveness of RANDOM SET, unless specified otherwise, we take the median out of a large number of runs (100 in our experiments).

HOT SET. Fuzz each seed for t seconds and return the top k seeds by number of unique bugs found. The rationale behind HOT SET is similar to multi-armed bandit algorithms—a buggy program is more likely to have more bugs. In our experiments, we fuzz each seeds for 5 minutes ($t = 300$) to compute the HOT SET.

UNWEIGHTED MINSET. Use an unweighted k-minset. This corresponds to standard coverage-based approaches [1, 23], and serves as a baseline for measuring their effectiveness. To compute UNWEIGHTED MINSET when k is greater than the minimum required to get full coverage, the minset is padded with files sorted based on the quality metric (coverage). We follow the same approach for TIME MINSET and SIZE MINSET.

TIME MINSET. Return a k-execution time weighted minset. This algorithm corresponds to Woo et al.'s observation that weighting by time in a multi-armed bandit fuzzing algorithm tends to perform better than the unweighted version [29]. The intuition is that seeds that are fast to execute ultimately lead to far more fuzz runs during a campaign, and thus potentially more bugs.

SIZE MINSET. Return a k-size weighted minset. Weighting by file size may change the ultimate minset, e.g., many smaller files that cover a few code blocks may be preferable to one very large file that covers many code blocks—both in terms of time to execute and bits to flip.

[2] Other algorithms exist to compute the weighted minset (see [6, 35-3.3]).

[3] This is a high-level abstraction of the `Delta` and `RunCoverage` methods. We checked the Peach implementation after the paper submission, and noticed that the sorting was removed (At Line 4 of the algorithm) in their MinSet implementation since Peach 3.1.95.

For example, SIZE MINSET will always select a 1KB seed over a 100MB seed, all other things being equal.

2.2 Specific Research Questions

Previous wisdom has suggested using UNWEIGHTED MINSET as the algorithm of choice for computing minsets [1, 23]. Is this justified? Further, computing the minset requires measuring code coverage. This computation requires time, time that could be spent fuzzing as in the HOT SET algorithm. Are coverage-based minsets beneficial and when?

More precisely, we formulate the following hypothesis:

Hypothesis 1 (MINSET > RANDOM.) *Given the same size parameter k,* MINSET *algorithms find more bugs than* RANDOM SET.

Hypothesis 1 is testing whether the heuristics applied by the algorithms presented above (§ 2.1) are useful. If they are as useful as choosing a random set, then the entire idea of using any of these MINSET algorithms is fundamentally flawed.

Hypothesis 2 (MINSET Benefits > Cost.) *Computing the* MINSET *for a given application and set of seed files and then starting fuzzing finds more bugs than just fuzzing.*

Hypothesis 2 tests whether the benefits of the minset outweigh the cost. Instead of spending time computing code coverage of seed files, should we instead spend it fuzzing? If yes, then the idea of reducing the files for *every* fuzzed application is flawed. It would also imply that precomputing minsets is necessary for the minsets to be useful. This observation leads to our next hypothesis.

Hypothesis 3 (MINSET Transferability.) *Given applications A and B that accept the same filetype F,* MINSET_F^A *finds the same or more bugs in application B as* MINSET_F^B.

Hypothesis 3 tests the transferability of seeds across applications that accept the same file type. For example, is the MINSET for PDF viewer A effective on PDF viewer B? If yes, we only need to compute a minset once per file type, thus saving resources (even if Hypothesis 2 is false).

Hypothesis 4 (MINSET Data Reduction.) *Given a target application A, a set of seed files F, and a* MINSET_F^A*, fuzzing with* MINSET_F^A *finds more bugs than fuzzing with the entire set F.*

Hypothesis 4 tests the main premise of using a reduced data set. Our MINSET contains fewer bugs than the full set in total. Under what conditions is the reduction beneficial?

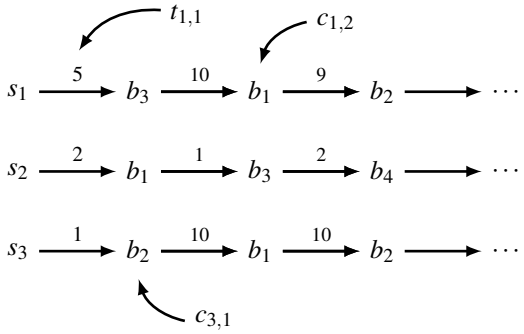

Figure 2: An example of output from fuzzing 3 seeds. Bugs may be found across seeds (e.g., b_1 is found by all seeds. A single seed may produce the same bug multiple times, e.g., with s_3. We also show the corresponding ILP variables t (interarrival times) and c (crash ids).

3 Q2: Measuring Selection Quality

There are a variety of seed selection strategies, e.g., to use minset or pick k seeds at random. How can we argue a particular seed selection strategy performs well?

One strawman answer is to run seed selection algorithm A to pick subset S_A, algorithm B to pick subset S_B. We then fuzz S_A and S_B for an equal amount of time and declare the fuzz campaign with the most bugs the winner. The fuzz campaign will incrementally fuzz each seed in each set according to its own scheduling algorithm. While such an approach may find the best seed selection for a *particular* fuzzing strategy, it provides no evidence that a particular subset is inherently better than another in the limit.

The main intuition in our approach is to measure the *optimal case* for bugs found with a particular subset of seeds. The best case provides an upper bound on any scheduling algorithm instead of on a particular scheduling algorithm. Note the lower bound on the number of bugs found for a subset is trivially zero, thus all we need is an upper bound.

To calculate the optimal case, we fuzz each seed in s_i for t seconds, recording as we fuzz the arrival rate of bugs. Given n seeds, the total amount of time fuzzing is $n * t$. For example, given 3 seeds we may have a bug b_i arrival time given by Figure 2.

Post-fuzzing, we then calculate the ex post facto optimal search strategy to maximize the number of bugs found. It may seem strange at first to calculate the optimal seed selection strategy after all seeds have been fuzzed at first blush. However, by doing so we can measure the quality of the seed selection strategy with respect to the optimal, thus give the desired upper bound. For example, if the seed selection strategy picks s_1 and s_2, we can calculate the maximum number of bugs that could be

found by any scheduler, and similarly for the set s_2, s_3 or any other set. Note we are calculating the upper bound to scientifically justify a particular strategy. For example, our experiments suggest to use UNWEIGHTED MINSET for seed selection. During a practical fuzzing campaign, one would not recompute the upper bound for the new dataset; instead, she would use the seed selection strategy that was shown to empirically perform best in previous tests.

3.1 Formalization

Let $Fuzz$ be a single-threaded fuzzer that takes in a set of seeds $\mathbb{C} \subseteq \mathbb{F}$ and a time threshold t_{thres} and outputs a sequence of unique bugs b_i along with the seed files that triggered them S_i and timestamps t_i:

$$Fuzz(\mathbb{C}, t_{\text{thres}}) = \{(b_1, S_1, t_1), \ldots, (b_n, S_n, t_n)\}$$

Given that we know the ground truth, i.e., we know the value of $Fuzz$ when applied on every singleton in $\mathbb{F}$: $Fuzz(\{S_i\}, t_{\text{thres}}) = \{(b_1, S_i, t_1), \ldots, (b_k, S_i, t_k)\}$, we can model the computation of the optimal scheduling/seed selection across all seed files in $\mathbb{F}$. Note that the ground truth is necessary, since any optimal solution can be only computed in retrospect (if we know how each seed would perform). We measure optimality of a scheduling/seed selection by computing the maximum number of unique bugs found.

The *optimal budgeted ex post facto scheduling problem* is given the ground truth for a set of seeds $Fuzz(\{S_i\}, t_{\text{thres}}) = \{(b_1, S_i, t_1), \ldots, (b_k, S_i, t_k)\}$ and a time threshold t_{thres}, automatically compute the interleaving of fuzzed seeds (time slice spent analyzing each one) to maximize the number of bugs found. The number of bugs found for a given minset gives an upper bound on the performance of the set and can be used as a quality indicator. Note that the same bug may be found by different seeds and may take different amounts of time to find.

Finding an optimal schedule for a given ground truth is currently an open problem. Woo et al. come the closest, but their algorithm assumes each seed produces independent bugs [29]. We observe finding an optimal scheduling algorithm is inherently an integer programming problem. We formulate finding the exact optimal seed scheduling as an Integer Linear Programming (ILP) problem. While computing the optimal schedule is NP-hard, ILP formulations tend to work well in practice.

First, we create an indicator variable for unique bugs found during fuzzing.

$$b_x = \begin{cases} 1 & \text{The schedule includes finding unique bug } x \\ 0 & \text{Otherwise} \end{cases}$$

The goal of the optimal schedule is to maximize the number of bugs. However, we do not see bugs, we see individual crashes arriving during fuzzing. We create an indicator variable $c_{i,j}$ that determines whether the optimal schedule includes the j^{th} crash of seed i:

$$c_{i,j} = \begin{cases} 1 & \text{The schedule includes crash } j \text{ for seed } i \\ 0 & \text{otherwise} \end{cases}$$

Note that multiple crashes $c_{i,j}$ may correspond to the same bug. Crashes are triaged to unique bugs via a uniqueness function denoted by μ. In our experiments, we use stack hash [24], a non-perfect but industry standard method. Thus, if the total number of unique stack hashes is U, we say we found U unique bugs in total. The invariant is:

$$b_x = 1 \text{ iff } \exists\, i, j : \mu(c_{i,j}) = x \tag{1}$$

Thus, if two crashes $c_{i,j}$ and $c_{i',j'}$ have the same hash, a schedule can get at most one unique bug by including either or both crashes.

Finally, we include a cost for finding each bug. We associate with each crash the incremental fuzzing cost for seed S_i to find the bug:

$$\forall i : t_{i,j} = \begin{cases} a_{i,1} & , j = 1 \\ a_{i,j} - a_{i,j-1} & , j > 1 \end{cases}$$

where $a_{i,j}$ is the arrival time for the $c_{i,j}$ crash, and $t_{i,j}$ represents interarrival time—the time interval between the occurrences of $c_{i,j-1}$ and $c_{i,j}$. Figure 2 visually illustrates the connection between $c_{i,j}$, b_x and $t_{i,j}$.

We are now ready to phrase optimal scheduling with a fixed time-budget as an ILP maximization problem:

$$\begin{aligned}
\text{maximize} \quad & \sum_x b_x \\
\text{subject to} \quad & \bigwedge_{i,j} . \, c_{i,j+1} \leq c_{i,j} && (2) \\
& \sum_{i,j} c_{i,j} \cdot t_{i,j} \leq t_{\text{thres}} && (3) \\
& \bigwedge_{i,j} . \, c_{i,j} \leq b_x \text{ where } \mu(c_{i,j}) = x && (4) \\
& \bigwedge_{x} . \, b_x \leq \sum_{i,j} c_{i,j} \text{ where } \mu(c_{i,j}) = x && (5)
\end{aligned}$$

Constraint (2) ensures that the schedule considers the order of crashes found. In particular, if the j-th crash of a seed is found, all the previous crashes must be found as well. Constraint (3) ensures that the time to find all the crashes does not exceed our time budget t_{thres}. Constraints (4) and (5) link crashes and unique bugs. Constraints (4) says that if a crash is found, its corresponding bug (based on stack-hash) is found, and the next equation guarantees that if a bug is found, at least one crash triggering this bug was found.

Additionally, by imposing one extra inequality:

$$\sum_i c_{i,1} \leq k \tag{6}$$

we can bound the number of used seeds by k (if the first crash of a seed is not found, there is no value in fuzzing the seed at all), thus getting k-bounded optimal budgeted scheduling, which gives us the number of bugs found with the optimal minset of size up to k.

Optimal Seed Selection for Round-Robin. The formulation for optimal budgeted scheduling gives us a best solution any scheduling algorithm could hope to achieve both in terms of seeds to select (minset) and interleaving between explored seeds (scheduling). We can also model the optimal seed selection for specific scheduling algorithms with the ILP formulation. We show below how this can be achieved for Round-Robin, as this may be of independent interest.

Round-Robin scheduling splits the time budget between the seeds equally. Given a time threshold t_{thres} and N seeds, each seed will be fuzzed for $\frac{t_{\text{thres}}}{N}$ units of time. Round-Robin is a simple but effective scheduling algorithm in many adversarial scenarios [29]. Simulating Round-Robin for a given set of seeds is straightforward, but computing the optimal subset of seeds of size k with Round-Robin cannot be solved with a polynomial algorithm. To obtain the optimal minset for Round-Robin, we add the following inequality to Inequalities 2-6:

$$\bigvee_i . \sum_j c_{i,j} \cdot t_{i,j} \leq \frac{t_{\text{thres}}}{k} \tag{7}$$

The above inequality ensures that none of the seeds will be explored for more than $\frac{t_{\text{thres}}}{k}$ time units, thus guaranteeing that our solution will satisfy the Round-Robin constraints. Similar extensions can be used to obtain optimal minsets for other scheduling algorithms.

4 Q3: Transferability of Seed Files

Precomputing a good seed set for a single application P_1 may be time intensive. For example, the first step in a minset-based approach is to run each seed dynamically to collect coverage information. Collecting this information may not be cheap. Collecting coverage data often requires running the program in a dynamic analysis environment like PIN [18] or Valgrind [25], which can slow down execution by several orders of magnitude. In our own experiments, collecting coverage information on our data set took 7 hours. One way COVERSET could minimize overall cost is to find a "good" set of seeds and reuse them from one application to another.

There are several reasons to believe this may work. One reason is most programs rely on only a few libraries for PDF, image, and text processing. For example, if application P_1 and P_2 both link against the `poppler` PDF library, both applications will likely crash on the same inputs. However, shared libraries are typically easy to detect, and such cases may be uninteresting. Suppose instead P_1 and P_2 both have independent implementations, e.g., P_1 uses `poppler` and P_2 uses the GhostScript graphics library. One reason P_1 and P_2 may crash on similar PDFs is that there are intrinsically hard portions of the PDF standard to implement right, thus both are likely to get it wrong. However, one could speculate any number of reasons the bugs in applications would be independent. To the best of our knowledge, there has been no previous systematic investigation to resolve this question when the bugs are found via fuzzing.

5 System Design

A precondition to fuzzing is configuring the fuzzer to take the seed file as input. In this step, we are given the entire database of seeds and a particular program under test P. To fuzz, we must:

1. Recover P's command line options.
2. Determine which argument(s) causes P to read in from the fuzzing source. For simplicity, we focus on reading from a file, but the general approach may work with other fuzzing sources.
3. Determine the proper file type, e.g., giving a PDF reader a PDF as a seed is likely to work better than giving a GIF. We say a file type is valid for a program if it does non-trivial processing on the file.

Current fuzz campaigns typically require a human to specify the above values. For our work, we propose a set of heuristics to help automate the above procedure.

In our approach, we first use simple heuristics to infer likely command lines. The heuristics try the obvious and common command line arguments for accepting files, e.g., `-f file`. We also brute force common help options (e.g., `-help`) and parse the output for additional possible command line arguments.

As we recover command lines, we check if they cause P to read from a file as follows. We create a unique file name x, run P `x`, and monitor for system calls that open `x`.

In our data set we have 274 different file types, such as JPEG, GIF, PNG, and video files. Once we know the proper way to run P with seeds, the question becomes which seed types should we give P? We infer appropriate file types based on the following hypothesis: if s is a seed

handled by P and s' is not, then we expect the coverage of $P(s) > P(s')$. This hypothesis suggests an efficient way to infer file types accepted by an application. First, create a set of sample file type seeds F, where each element consists of a seed for a unique file type. Second, for each $s_i \in F$, count the number of basic blocks executed by $P(s_i)$. Third, select the top candidate (or candidates if desired) by total execution blocks. Though simple, we show in our evaluation this strategy works well in practice.

6 Experiments

We now present our experiments for checking the validity of the hypotheses introduced in § 2.2 and evaluate the overall performance in terms of bug discovered of CovERset. We start by describing our experimental setup.

Experimental Setup. All of our experiments were run on medium and small VM instance types on Amazon EC2 (the type of the instance used is mentioned in every experiment). All VMs were running the same operating system, Debian Linux 7.4. The fuzzer used throughout our experiments is the CERT Basic Fuzzing Framework (BFF) [15]. All seed files gathered for our fuzzing experiments (4,912,142 files making up more than 6TB of data) were automatically crawled from the internet using the Bing API. Specifically, file type information was extracted from the open source Gnome Desktop application launcher data files and passed to the Bing API such that files of each type could be downloaded, filtered, and stored on Amazon S3. Coverage data was gathered by instrumenting applications using the Intel PIN framework and a standard block-based coverage collection PIN tool.

6.1 Establishing Ground Truth

To test the MinSet hypotheses, we need to obtain the ground truth (recall from § 3.1) for a fuzzing campaign that accounts for every possible seed selection and scheduling. We now present our methodology for selecting the target applications, files to fuzz, and parameters for computing the ground truth.

Target Applications. We selected 10 applications and 5 popular file formats: PDF, MP3, GIF, JPG and PNG for our experiments. Our program selection contains GUI and command line applications, media viewers, players, and converters. We manually mapped each program to a file format it accepts and formed 13 distinct (application, file formats) to be fuzzed—shown in Table 2. We selected at least two distinct command lines for each file type to test transferability (Hypothesis 3).

Seed Files. For each file type used by the target applications, we sampled uniformly at random 100 seed files (hence selecting $|\mathbb{F}| = 100$ for the seed file pool size) of the corresponding type from our seed file database. Note that determining ground truth for a single seed requires 12 hours, thus finding ground truth on all 4,912,142 is—for our resources—infeasible.

Fuzzing Parameters. Each of the target applications was fuzzed for 12 hours with each of the 100 randomly selected seed files of the right file type. Thus, each target application was fuzzed for 1,200 hours for a total of 650 CPU-days on an EC2 (m1.small) instance. All detected crashes were logged with timestamps and triaged based on BFF's stack hash algorithm.

The end result of our ground truth experiment is a log of crashes for each (seed file, application) tuple:

$$BFF(\{S_i\}, t_{\text{thres}} = 12h) = \{(b_1, S_i, t_1), \ldots, (b_k, S_i, t_k)\}$$

Fuzzing results. BFF found 2,941 unique crashes, identified by their stack hash. BFF crashed 8 programs out of the 10 target applications. 2,702 of the unique crashes were found on one application, `mp3gain`. Manual inspection showed that the crashes were due to a single exploitable buffer overflow vulnerability that mangled the stack and confused BFF's stack-based uniqueness algorithm. When reporting our results, we therefore count the 2,702 unique crashes in `mp3gain` as one. With that adjustment, BFF found 240 bugs. Developing and experimenting with more robust, effective, and accurate triaging algorithms is an open research problem and a possible direction for future work.

Simulation. The parameters of the experiment allow us to run simulations and reason about all possible seed selections (among the 100 seeds of the application) and scheduling algorithms for a horizon of 12 hours on a single CPU. Our simulator uses our ILP formulation from § 3 to compute optimal seed selections and scheduling for a given time budget. Using the ground truth, we can run simulations to evaluate the performance of hour-long fuzzing campaigns within minutes, following a replay-based fuzzing simulation strategy similar to FuzzSim [29].

We used the simulator and ran a set of experiments to answer the following three questions: 1) how good are seed selection algorithms when compared against Random Set (§ 6.2) and when compared against each other (§ 6.2.1)?, 2) can we reuse reduced sets across programs (§ 6.3)?, and 3) can our algorithm correctly identify file types for applications (§ 6.4)?

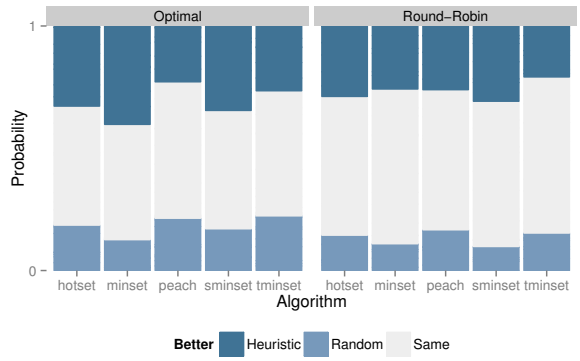

Figure 3: Comparing bug-finding performance of seed selection algorithms against RANDOM SET.

	Optimal	Round-Robin
HOT SET	63.58%	67.12%
PEACH SET	50.64%	60.30%
UNWEIGHTED MINSET	75.24%	70.24%
SIZE MINSET	66.33%	75.78%
TIME MINSET	52.60%	57.62%

Table 1: Conditional probability of an algorithm outperforming RANDOM SET with k=10, given that they do not have the same performance (P_{win}).

6.2 Are Seed Selection Algorithms Better than Random Sampling?

Spending resources on a seed selection algorithm is only useful if the selected seeds outperform random seed sampling (RANDOM SET). In this experiment, we compare the performance of selection algorithms as presented in §2.1 against the random sampling baseline.

All selection algorithms are deterministic, while RANDOM SET is randomized. Thus, we cannot show that RANDOM SET is always better (or worse), but we can instead compute the probability that RANDOM SET is better (or worse). To estimate the probability, we setup the following random experiment: we randomly sample a set of seeds—the size of the set is the same ($k = 10$ in our experiment for an order of magnitude reduction) as the competing reduced set—from the seed pool and measure the number of bugs found. The experiment has three possible outcomes: 1) the random set finds more bugs, 2) the random set finds fewer bugs, or 3) the random and the competitive set find the same number of bugs.

We performed 13,000 repetitions of the above experiment—1,000 for each (application, file format) tuple—and measured the frequency of each event when the optimal scheduling algorithm is employed for both. We then repeated the same experiment while using Round-Robin as the scheduling algorithm. We calculated the probability by dividing the frequency by the number of samples. Figure 3 summarizes the results. For instance, the left-most bar is the result for HOT SET with the optimal scheduling. You can see that HOT SET finds more bugs than a RANDOM SET of the same size with a probability of 32.76%, and it is worse with a probability of 18.57%. They find the same amount of bugs with a probability of 48.66%.

The first pattern that seems to persist through scheduling and selection algorithms (based on Figure 3) is that there is a substantial number of ties—RANDOM SET seems to behave as well as selection algorithms for the majority of the experiments. This is not surprising, since 3/13 (23%) of our (application, file format) combinations—(mplayer, MP3), (eog, JPG), (jpegtran, JPG)—do not crash at all. With no crash to find, any algorithm will be as good as random. Thus, to compare an algorithm to RANDOM SET we focus on the cases where the two algorithms differ, i.e., we compute the conditional probability of winning when the two algorithms are not finding the same number of bugs.

We use P_{win} to denote the conditional probability of an algorithm outperforming RANDOM SET, given that they do not have the same performance. For example, for SIZE MINSET, P_{win} is defined as: $P[\text{SIZE MINSET} > \text{RANDOM SET} \mid \text{SIZE MINSET} \neq \text{RANDOM SET}]$. Table 1 shows the values of P_{win} for all algorithms for sets of size $k = 10$. We see that UNWEIGHTED MINSET and SIZE MINSET are the algorithms that more consistently outperform RANDOM SET with a P_{win} ranging from 66.33% to 75.78%. HOT SET immediately follows in the 63-67% range, and TIME MINSET, PEACH SET have the worst performance. Note that PEACH SET has a P_{win} of 50.64% in the optimal schedule effectively meaning that it performs very close to a random sample on our dataset.

Conclusion: seed selection algorithms help. With the exception of the PEACH SET and TIME MINSET algorithms which perform very close to RANDOM SET, our data shows that heuristics employed by seed selection algorithms perform better than fully random sampling. Thus, hypothesis 1 seems to hold. However, the bug difference is not sufficient to show that *any* of the selection algorithms is *strictly* better with statistical significance. Fuzzing for longer and/or obtaining the ground truth for a larger seed pool are possible future directions for showing that seed selection algorithms are *strictly* better than choosing at random.

Files	Programs	Crashes	Bugs	RANDOM SET		HOT SET		UNWEIGHTED MINSET		TIME MINSET		SIZE MINSET		PEACH SET	
				#S	#B	#S	#B	#S	#B	#S	#B	#S	#B	#S	#B
PDF	xpdf	706	57	10	7	10	9	32	19	32	16	40	19	54	31
	mupdf	6,570	88	10	13	10	14	40	29	43	29	49	31	59	31
	pdf2svg	5,720	81	10	14	10	27	36	48	39	43	45	47	53	49
MP3	ffmpeg	1	1	10	0	10	1	11	0	11	0	22	0	19	0
	mplayer	0	0	10	0	10	0	10	0	12	0	14	0	23	0
	mp3gain	434,400	2,702	10	92	10	9	9	150	8	74	10	74	14	175
GIF	eog	9	1	10	0	10	1	29	0	27	0	43	1	44	1
	convert	72	2	10	1	10	1	13	1	14	0	24	2	22	1
	gif2png	162,302	6	10	4	10	4	16	5	17	5	29	5	33	4
JPG	eog	0	0	10	0	10	0	31	0	31	0	47	0	53	0
	jpegtran	0	0	10	0	10	0	10	0	12	0	21	0	23	0
PNG	eog	123	2	10	1	10	1	30	2	30	2	45	2	49	2
	convert	2	1	10	0	10	0	11	1	12	1	17	1	16	1
Total		609,905	2,941		132		67	278	255	288	170	406	182	462	295

Table 2: Programs fuzzed to evaluate seed selection strategies and obtain ground truth. The columns include the number of seed files (#S) obtained with each algorithm, and the number of bugs found (#B) with the optimal scheduling strategy.

6.2.1 Which Algorithm Performed Best?

Table 2 shows the full breakdown of the reduced sets computed by each algorithm with the optimal scheduling algorithm. Columns 1 and 2 show the file type and program we are analyzing, while columns 3 and 4 show the total number of crashes and unique bugs (identified by stack hash) found during the ground truth experiment. The next six columns show two main statistics (in subcolumns) for each of the seed selection algorithms: 1) the size of the set k (#S), and 2) the number of bugs (#B) identified with optimal scheduling. All set cover algorithms (PEACH SET, UNWEIGHTED MINSET, TIME MINSET, SIZE MINSET) were allowed to compute a full-cover, i.e., select as many files as required to cover all blocks. The other two algorithms (RANDOM SET and HOT SET) were restricted to sets of size $k = 10$.

Bug Distribution and Exploitability. The fuzzing campaign found bugs in 10/13 configurations of ⟨program, file type⟩, as shown in table 2. In 9/10 configurations we found less than 100 bugs, with one exception: `mp3gain`. We investigated the outlier further, and discovered that our fuzzing campaign identified an exploitable stack overflow vulnerability—the mangled stack trace can create duplicates in the stack hash algorithm. We verified the bug is exploitable and notified the developers, who promptly fixed the issue.

Reduced Set Size. Table 2 reflects the ability of the set cover algorithms to reduce the original dataset of 100 files. As expected, UNWEIGHTED MINSET is the best in terms of reduction ability, with 278 files for obtaining full cover. TIME MINSET requires slightly more files (288). SIZE MINSET and PEACH SET require almost twice as many files to obtain full cover (406 and 462 respectively).

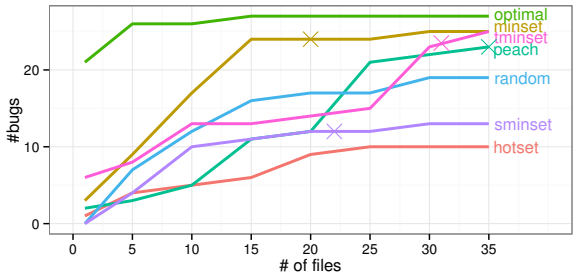

Figure 4: Number of bugs found by different seed selection algorithms with optimal scheduling.

Bug Finding. The PEACH SET algorithm finds the highest number of bugs (295), followed by UNWEIGHTED MINSET (255), SIZE MINSET (182) and TIME MINSET (170). HOT SET and RANDOM SET find substantially fewer bugs when restricted to subsets of size up to 10. We emphasize again that bug counts are measured under optimal scheduling and thus size of the reduced set is analogous to the performance of the selection algorithm (the highest number of bugs will be found when all seeds are selected). Thus, to compare sets of seeds in terms of bug-finding ability we need a head to head comparison where sets have the same size k.

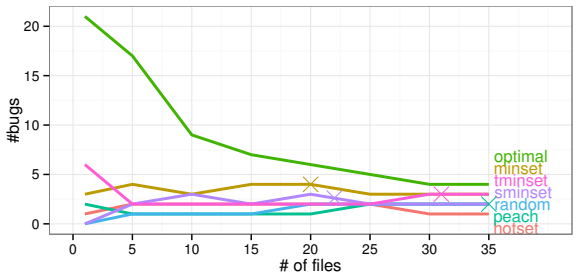

Figure 5: Number of bugs found by different seed selection algorithms with Round-Robin.

Figure 4 shows all selection algorithms and how they perform in terms of average number of bugs found as a function of the parameter k—the size of the seed file set. The "×" symbols represent the size after which each algorithm achieves a full cover (after that point extra files are added sorted by the metric of the selection algorithm, e,g,, by coverage in UNWEIGHTED MINSET). As witnessed in the comparison against RANDOM SET, UNWEIGHTED MINSET consistently performs better than other seed selection algorithms. TIME MINSET and PEACH SET also eventually converge to the performance of UNWEIGHTED MINSET under optimal scheduling, closely followed by random. HOT SET performs the worst, showing that spending time exploring *all* seeds can be wasteful. We also note, that after obtaining full cover (at 20 seed files), UNWEIGHTED MINSET's performance does not improve at the same rate—showing that adding new files that do not increase code coverage is not beneficial (even with optimal scheduling).

We performed an additional simulation, where all reduced sets were run with Round-Robin as the scheduling algorithm. Figure 5 shows the performance of each algorithm as a function of the parameter k. Again, we notice that that UNWEIGHTED MINSET is outperforming the other algorithms. More interestingly, we also note that UNWEIGHTED MINSET's performance actually *drops* after obtaining full cover. This shows that minimizing the number of seeds is important; adding more seeds in Round-Robin seems to hurt performance for all algorithms.

Conclusion: UNWEIGHTED MINSET performed best. UNWEIGHTED MINSET outperformed all other algorithms in our experiments, both for optimal and Round-Robin scheduling. This experiment confirms conventional wisdom that suggests collecting seeds with good coverage for successful fuzzing. More importantly, it also shows that computing a minimal cover with an approximation with a proven competitiveness ratio (UNWEIGHTED MINSET) is better than using an algorithm with no guaranteed competitive ratio (PEACH SET).

File	Application	FULL SET	UNWEIGHTED MINSET (k=10)
PDF	xpdf	53%	70%
	mupdf	83%	90%
	pdf2svg	71%	80%
MP3	ffmpeg	1%	0%
	mplayer	0%	0%
	mp3gain	95%	100%
GIF	eog	8%	0%
	convert	12%	10%
	gif2png	97%	100%
JPG	eog	0%	0%
	jpegtran	0%	0%
PNG	eog	22%	30%
	convert	2%	10%

Table 3: Probability that a seed will produce a bug in 12 hours of fuzzing.

6.2.2 Are reduced seed sets better than a full set?

Hypothesis 4 tests the premise of using a reduced data set. Will a reduced set of seeds find more bugs than the full set? We simulated a fuzzing campaign with the full set, and with different reduced sets. We compare the number of bugs found by each technique.

Using the optimal scheduling, the full set will *always* find more, or the same amount of bugs, than any subsets of seeds. Indeed, the potential schedules of the full set is a superset of the potentiel schedules of any reduced set. The optimal schedule of a reduced set of seeds is a valid schedule of the full set, but the optimal schedule of the full set might not be a valid schedule of a reduced set. Hypothesis 4 is therefore false under the optimal scheduling. We use a Round-Robin schedule to answer this question more realistically.

The "×" symbols on Figure 5 shows the unpadded size of the different selection algorithms. For those sizes, UNWEIGHTED MINSET found 4 bugs on average, and the other MINSET algorithms found between 2.5 and 3 bugs. Fuzzing with the full set uncovered only 1 unique bug on average.

We also measure the quality of a set of seeds by looking at the average seed quality contained in that set. Our hypothesis is that a reduced set increases the average seed quality compared to the full set. To measure quality, we computed the probability of a seed producing a bug after fuzzing it for 12 hours, when the seed is picked from the full set or the UNWEIGHTED MINSET. Table 3 lists the

results of this experiment. The UNWEIGHTED MINSET had a higher seed quality than the full set in 7 cases, while the opposite was true in 3 cases. They were tied on the 3 remaining cases.

Conclusion: Fuzzing with a reduced sets is more efficient in practice. The UNWEIGHTED MINSET outperformed the full set in our two experiments. Our data demonstrates that using seed selection techniques is beneficial to fuzzing campaigns.

6.3 Are Reduced Sets Reusable Across Programs?

We showed that seed selection algorithms improve fuzzing in terms of bug-finding performance. However, performing the data reduction may be computationally expensive; for instance, all set cover algorithms require collecting coverage information for all the seeds. Is it more profitable to invest time computing the minset to fuzz an efficient reduced set, or to simply fuzz the full set of seeds for the full time budget? In other words, is the seed selection worth the effort to be performed online?

We answer that question by presenting parts of our dataset. For example, our JPG bucket contains 530,727 distinct files crawled from the web. Our PIN tool requires 55 seconds (on average based on the 10 applications listed in Table 2) to compute code coverage for a single seed. Collecting coverage statistics for all our JPG files would take 368 CPU-days. For fuzzing campaigns shorter than a year, there would not be enough time to compute code coverage, let alone finding more bugs than the full set.

The result above indicates that, while seed selection techniques help improve the performance of fuzzing, their benefits may not outweigh the costs. It is impractical to spend a CPU year of computation to perform a separate seed selection for every new application that needs fuzzing, thus indicating that Hypothesis 2 does not hold.

However, recomputing the reduced set for *every application* may not be necessary. Instead, we can compute a reduced set for every *file type*. Our intuition is a reduced set that is of high-quality for application A should also be high-quality for application B—assuming they accept the same file type. Thus, precomputing reduced sets for popular file types *once*, would allow us to instantly select a high-quality set of seed files to start fuzzing. To test transferability of reduced sets (Hypothesis 3), we measure seed quality by computing code coverage achieved by a MINSET across programs.

Do Reduced Sets Transfer Coverage? Using the seed files from our ground truth experiment (§ 6.1) we measured the cumulative code coverage achieved in each configuration (program and file format) with reduced UNWEIGHTED MINSETs computed on all other configurations (for a total of $13 \times 13 \times 100$ coverage measurements). All measurements were performed on a c1.medium instance on amazon.

Figure 6 is a heat map summarizing our results. The configurations on the bottom (x-axis) represent all computed UNWEIGHTED MINSETs, while the configurations on the left (y-axis) represent the configurations tested. Darker colors indicate that the selected UNWEIGHTED MINSET obtains higher coverage. For example, if we select the `pdf.mupdf` MINSET from the x-axis, we can see how it performs on all the other configurations on the y-axis. For instance, we notice that `pdf.mupdf` MINSET performs noticeably better on 5 configurations: `pdf.mupdf` (expected since this is the configuration on which we computed the MINSET), `pdf.xpdf` and `pdf.pdf2svg` (expected since these applications also accept `pdfs`), and interestingly `png.convert` and `gif.convert`. Initially we were surprised that a PDF MINSET would perform so well on `convert`; it turns out that this result is not surprising since `convert` can also process PDF files. Similar patterns can be similarly explained—for example, GIF MINSETs are performing *better* than MP3 MINSETs for `mplayer`, simply because `mplayer` can render GIF images.

The heat map allows us to see two clear patterns:

1. High coverage indicates the application accepts a file type. For instance, by following the row of the `gif.eog` configuration we can immediately see that `eog` accepts GIF, JPG, and PNG files, while it does not process MP3s or PDFs. This is exactly the same pattern we are exploiting in our file type inference algorithm (§ 6.4).

2. Coverage transfers across applications that process the same file type. For example, we clearly see the PDF cluster forming across all PDF configurations, despite differences in implementations. While `xpdf` and `pdf2svg` both use the `poppler` library for processing PDFs, `mupdf` has a completely independent implementation. Nevertheless, `mupdf`'s MINSET performs well on `xpdf` and vice versa. Our data shows that similar clusters appear throughout configurations of the same file type, suggesting that we can reuse MINSETs across applications that accept the same file type (Hypothesis 3).

Conclusion: Reduced sets are transferable. Our data suggests that reduced sets can be transferred to programs parsing the same file types with respect to code coverage. Therefore, it is necessary to compute only one reduced set per file type.

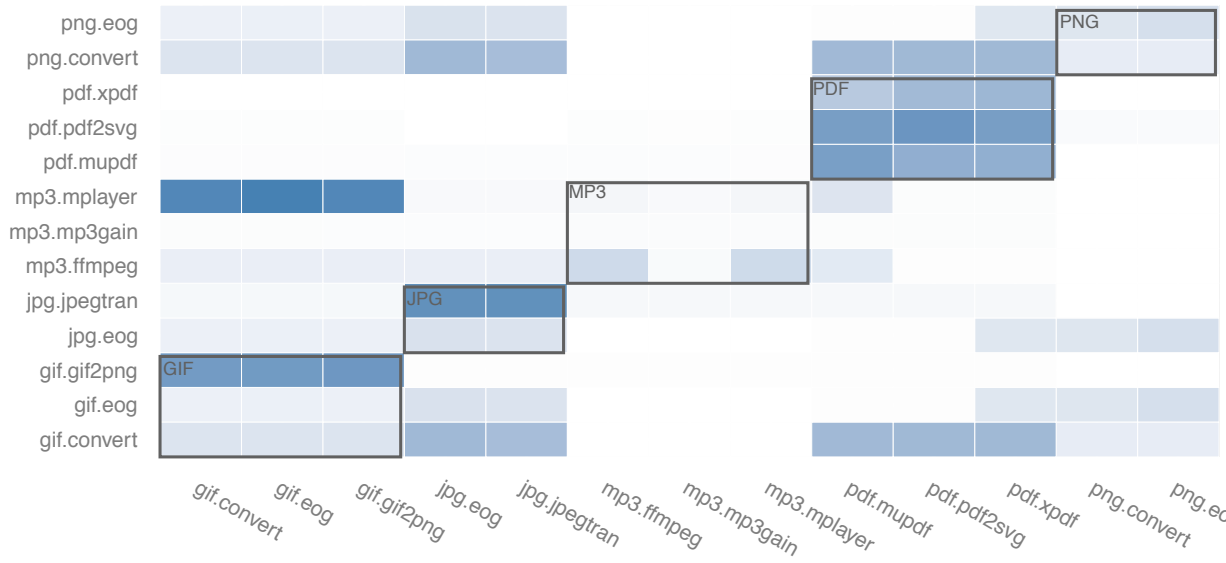

Figure 6: Transferability of UNWEIGHTED MINSET coverage across configurations. The base configurations, on which the reduced sets were computed, are on the bottom; the tested configurations are on the left. Darker colors indicate higher coverage.

6.4 Inferring File Types

In this experiment we ran our algorithm to automatically infer file types for our 10 applications over a large body of diverse file types (88 in total including AVI, MP3, MKV and so forth). We then manually verified the inferred file type, and measured accuracy. We report an error if a reported file type is not recognized by the target application. Table 6.4 summarizes our results. Our file type inference algorithm successfully infers file types for every program except `mp3gain`, where the file type inferred was CBR (Comic Book Reader), instead of MP3.

We manually examined why `mp3gain` shows higher code coverage for CBR files. It turns out that our sample CBR file is larger than 15 MB, and it happens to have a valid MP3 frame header signature in the middle of the file. Since `mp3gain` searches for a valid MP3 frame header regardless of the entire file format, it is possible to misinterpret an input file as a valid MP3 file. In other words, this is probably not a false positive, because `mp3gain` indeed takes in the CBR file and outputs a modified CBR file, which is the expected behavior of the program.

Program	Inferred File Type	Success
convert	svg	✓
eog	png	✓
ffmpeg	divx	✓
gif2png	gif	✓
jpegtran	jpeg	✓
mp3gain	cbr	✗
mplayer	avi	✓
mupdf	pdf	✓
pdf2svg	pdf	✓
xpdf	pdf	✓

Table 4: File-type inference results on our dataset.

7 Discussion & Future Work

Input Types. The focus of this paper is on file-parsing applications. Thus, we do not target applications that use command line arguments as their input sources (e.g., /bin/echo), or applications that receive input from the network (e.g., /usr/bin/wget). File-based vulnerabilities represent a significant attack vector, since remote attacks can be carried out by simply sending an attachment to the victim over the network.

Handling argument inputs for applications is straightforward: we can extend our fuzzing framework to randomly generate arguments to the `exec` system call. Treating network applications would be more elaborate since we would have to update our seed database to include network packets for various protocol types. One potential extension is to utilize automatic protocol reversing [3, 7]. We leave it as future work to support more input types.

Statistical Significance. We have performed initial hypothesis testing based on the data. Currently, using UNWEIGHTED MINSET is assumed to outperform other al-

gorithms. Using our data (see Figure 3), we were able to show that UNWEIGHTED MINSET is at least as good as random with high probability. However, the data does not show with statistical significance that UNWEIGHTED MINSET is strictly better. Fuzzing longer and with more seed files and programs may yield more datapoints that would allow a stronger conclusion—at the cost of a much more costly ground truth computation: an additional seed file requires 12 hours of extra fuzzing hours for each application. We leave fuzzing for more than 650 days as future work.

Command Line Inference. Currently, COVERSET uses several heuristics to infer command line arguments. We believe that command line inference is an important first step to fully automate the entire fuzzing process. For example, COVERSET currently does not handle dependent arguments, e.g., when option `A` is valid only when option `B` is also selected. Developing systematic and effective approaches for deriving command line arguments—e.g., based on white-box techniques—is a possible direction for future work.

8 Related Work

In early 90s, Miller et al. [21] introduced the term fuzzing. Since then, it has become one of the most widely-deployed technique for finding bugs. There are two major categories in fuzzing based on its ability to examine the internal of the software under test: (1) black-box fuzzing [20], and (2) white-box fuzzing [13, 14]. In this paper, we use black-box mutational fuzzing as the underlying technique for our data reduction algorithms.

Coverage-driven seed selection is not new. Several papers and security practitioners use similar heuristics to select seeds for fuzzing [1, 8, 9, 11, 22, 23, 26]. FuzzSim by Woo et al. [29] is the closest work from academia, where they tackle a seed scheduling problem using multi-armed bandit algorithms. Our paper differs from their approach in that we are not developing an online scheduling algorithm, but an offline data-driven seed selection approach. Therefore, our seed selection is complementary—when it is used as a preprocessing step—to the FuzzSim scheduling algorithm.

There are several previous works on recovering input formats, which involves dynamic taint analysis [3, 7]. Our goal is being able to run fuzzing with appropriate seed files, but not recovering the semantics of file format. Accommodating more precise file format inference techniques is out of the scope of this paper.

9 Conclusion

In this paper we designed and evaluated six seed selection techniques. In addition, we formulated the optimal ex post facto seed selection scheduling problem as an integer linear programming problem to measure the quality of seed selection algorithms. We performed over 650 days worth of fuzzing to determine ground truth values and evaluated each algorithm. We found 240 new bugs. Our results suggest how best to use seed selection algorithms to maximize the number of bugs found.

Acknowledgments

We would like to thank Alan Hall, and our anonymous reviewers for their comments and suggestions. This work is supported in part by the NSF-CNS0953751, DARPA CSSG-FA9750-10-C-0170, and the SEI-FA8721-05-C-0003. This work reflects only the opinions of the authors, not the sponsors.

References

[1] ABDELNUR, H., LUCANGELI, O., AND FESTOR, O. Spectral fuzzing: Evaluation & feedback.

[2] ARIANE. The ariane catastrophe. http://www.around.com/ariane.html.

[3] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer & Communications Security* (2007).

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

[5] CNN. Toyota recall costs: $2 billion. http://money.cnn.com/2010/02/04/news/companies/toyota_earnings.cnnw/index.htm, 2010.

[6] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms , Second Edition*, vol. 7. 2001.

[7] CUI, W., PEINADO, M., AND CHEN, K. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008).

[8] DURAN, D., WESTON, D., AND MILLER, M. Targeted taint driven fuzzing using software metrics. In *CanSecWest* (2011).

[9] EDDINGTON, M. Peach fuzzer. http://peachfuzzer.com/.

[10] FEIGE, U. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM 45*, 4 (1998), 634–652.

[11] FRENCH, T., AND PROJECT, V. Closed loop fuzzing algorithms.

[12] GARTNER. Software security market at 19.2 billion. http://www.gartner.com/newsroom/id/2500115, 2012.

[13] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium* (2008), no. July.

[14] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: White-box fuzzing for security testing. *Communications of the ACM 55*, 3 (2012), 40–44.

[15] HOUSEHOLDER, A. D., AND FOOTE, J. M. Probability-based parameter selection for black-box fuzz testing. Tech. Rep. August, CERT, 2012.

[16] JOHNSON, D. S. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences 9* (1974), 256–278.

[17] LANGNER, R. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy Magazine 9*, 3 (May 2011), 49–51.

[18] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tols with dynamic instrumentation. In *Programming Language Design and Implementation* (2005), ACM, pp. 190–200.

[19] MARKET RESEARCH MEDIA. U.s. federal cybersecurity market forecast 2013-2018. http://www.marketresearchmedia.com/?p=206, 2013.

[20] MCNALLY, R., YIU, K., AND GROVE, D. Fuzzing : The state of the art.

[21] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of unix utilities. *Communications of the ACM 33*, 12 (1990), 32–44.

[22] MILLER, C. Fuzz by number. In *CanSecWest* (2008).

[23] MILLER, C. Babysitting an army of monkeys. In *CanSecWest* (2010).

[24] MOLNAR, D., LI, X., AND WAGNER, D. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium* (2009), pp. 67–82.

[25] NETHERCOTE, N., AND SEWARD, J. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science 89*, 2 (Oct. 2003), 44–66.

[26] OPSTAD, L., AND MOLNAR, D. Effective fuzzing strategies. Tech. rep., 2010.

[27] TEAM, C. S. Clusterfuzz. https://code.google.com/p/clusterfuzz/.

[28] UHLEY, P. A basic distributed fuzzing framework for foe. https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html.

[29] WOO, M., CHA, S. K., GOTTLIEB, S., AND BRUMLEY, D. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM Conference on Computer & Communications Security* (2013), pp. 511–522.