Cerebro: Context-aware Adaptive Fuzzing for Effective Vulnerability Detection

Anonymous Author(s)

ABSTRACT

Existing greybox fuzzers mainly utilize program coverage as the goal to guide the fuzzing process. To maximize their outputs, coverage-based greybox fuzzers need to evaluate the quality of seeds properly, which involves making two decisions: 1) which is the most promising seed to fuzz next (seed prioritization), and 2) how many efforts should be made to the current seed (power scheduling). In this paper, we present our fuzzer, Cerebro, to address the above challenges. For the seed prioritization problem, we propose an online multi-objective based algorithm to balance various metrics such as code complexity, coverage, execution time, etc. To address the power scheduling problem, we introduce the concept of input potential to measure the complexity of uncovered code and propose a cost-effective algorithm to update it dynamically. Unlike previous approaches where the fuzzer evaluates an input solely based on the execution traces that it has covered, Cerebro is able to foresee the benefits of fuzzing the input by adaptively evaluating its input potential. We perform a thorough evaluation for Cerebro on 6 different real-world programs. The experiments show that Cerebro can find more vulnerabilities and achieve better coverage than state-of-the-art fuzzers such as AFL and AFLFast. Additionally, we found 15 previously unknown bugs in mjs (a light-weight Javascript engine for embedded systems), Intel XED (Intel X86 Encoder Decoder) during the experiments and 1 new CVE in Radare2 (a popular reverse engineering framework). Furthermore, all the new bugs are confirmed by the developers and fixed.

1 INTRODUCTION

Fuzzing, or fuzz testing, is progressively gaining popularity in both industry and academia since proposed decades before [1]. Various fuzzing tools (fuzzers) have been springing up to fulfill different testing scenarios in recent years [2]. These fuzzers can be classified as blackbox, whitebox, and greybox based on the awareness of the structural information about the program under test (PUT). Blackbox fuzzers [3] have no knowledge about the internals of PUT. So they can scale up but may not be effective. On the contrary, whitebox fuzzers utilize heavy-weight program analysis techniques (e.g. symbolic execution tree [4]) to improve effectiveness at the cost of scalability. To have the best of both worlds, greybox fuzzers (GBFs), such as AFL [5], are advocated to achieve scalability yet effectiveness. Fig. 1 depicts the workflow of greybox fuzzing.

A recent trend in academia is to make greybox fuzzing whiter with various light-weight program analysis. For example, Vuzzer [6], Steelix [7], Angora [8] and T-Fuzz [9] mainly help GBFs to penetrate path constraints via modifications on the *seed mutator* and *feedback collector* modules in Fig. 1. However, based on the nature that fuzzing's results are strong related with the seeds[1], the effects of all the works on these modules can be further maximized by enhancing the seeds' quality. To be more specific, as shown in the Fig. 1, GBFs need to deal with two fundamental problems — how to select the next seed to fuzz (***seed prioritization***) and how many new inputs need to be generated with the selected seed (***power scheduling***).

In most GBFs, new inputs are generated by mutating the seeds. If an input exercises a new path, it is retained by the GBF as a new seed; otherwise, it is discarded. In such a manner, the GBF maintains an increasing set of seeds. To maximize the number of bugs detected within a limited time budget, the GBF needs to wisely put the seeds in order by prioritizing the seeds with better quality. We call this ***seed prioritization*** problem. After seed prioritization, the GBF needs to decide, for each seed, the number of new inputs (a.k.a. "energy" in AFLFast [10]) to be generated from that seed. Ideally, the GBF should allocate more energy to a seed that brings more benefits via mutations. We call this ***power scheduling*** problem.

Solving both problems requires evaluations for the quality of the seeds using the (runtime) information of target program. Most existing GBFs evaluate a seed with knowledge extracted from its execution trace and other information such as file size and execution time. Some research endeavors have been made on the two problems [6, 10, 11]. Among these studies, Shudrak *et al.* [11] propose to use software complexity to facilitate the seed prioritization for fuzzers based on the assumption that complex code is more error prone. In AFLFast [10], Böhme *et al.* address both problems by prioritizing the seeds that exercise rarely executed paths, in hope that fuzzing such seeds can achieve more coverage rapidly. Despite these efforts, two challenges remain to be addressed. The first challenge is — whatever information (paths rarity or code complexity) about the execution traces are utilized, existing GBFs are not aware of the uncovered code close to the execution trace (i.e. context). Lacking such knowledge is because that context awareness normally requires heavy-weight program analysis techniques, which can hinder the performance of GBFs. However, context awareness can be very helpful in the fuzzing process, e.g., if the neighboring code around an execution trace gets covered, then mutating the seed holding this trace becomes less beneficial as the potential of leading to new coverage has dropped. The second challenge is — existing fuzzers either utilize a single-objective for seed prioritization [11] or mix several objectives via linear scalarization into one single-objective (a.k.a. weighted-sum) [5] [10] to perform seed prioritization. On one hand, using one single objective may cause bias and starve certain seeds. On the other hand, the weights used in linear scalarization are empirically decided without statistical justifications.

In this paper, we propose Cerebro to make proper decisions for seed prioritization and power scheduling by addressing the above challenges. Cerebro focuses on the *seed evaluator* and *seed queue* in Fig. 1. To bring context awareness to GBFs, we propose a new definition, named **input potential** — the complexity of not yet

[1]In this paper, we denote *all* the files fed to the PUT by fuzzers as *inputs*, and only those inputs kept by fuzzers for subsequent mutations as *seeds*.

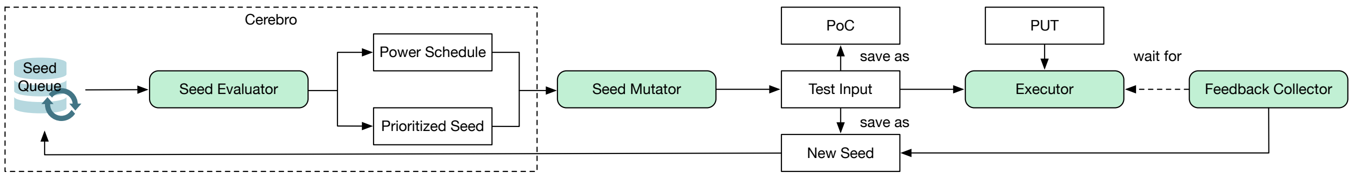


Figure 1: The workflow of greybox fuzzing

covered code near the execution trace of the input, together with an online algorithm to efficiently calculate the potentials of inputs with the ever changing context. To balance multiple important but conflicting objectives, we propose a multi-objective optimization (MOO) model together with a *nondominated sorting* based algorithm to quickly calculate the *Pareto Frontier* for a growing set of seeds. Technically, in CEREBRO, the MOO model is applied to perform seed prioritization and input potential is applied to facilitate power scheduling.

We implement CEREBRO and evaluate it with 8 widely-used real-world programs from different projects. The experiments scale up to more than 300 CPU-days. CEREBRO outperforms AFLFAST and AFL with significantly improved bug detection capability while maintaining a good coverage. Moreover, we find 15 previously unknown bugs in *mjs* and *xed*[2], and 1 CVE in *radare2*. Besides, all the new bugs have been confirmed and fixed.

The contributions of the paper are as following:

- We formulate a new concept, input potential, which represents the complexity of uncovered code close to an execution trace. Together with the concept, we also propose a cost-effective algorithm to quickly calculate the input potential to facilitate power scheduling.
- We propose a multi-objective based model together with an efficient nondominated sorting based algorithm for seed prioritization.
- We implement CEREBRO and evaluate its effectiveness with experiments from several aspects. The results are promising as we discover 15 previously unknown bugs and 1 CVE in widely-used open-source projects and all these bugs are confirmed and fixed.

2 BACKGROUND & PROBLEM STATEMENT

In this section, we first introduce the background of greybox fuzzing and then define the two research problems addressed in CEREBRO.

2.1 Background of Greybox Fuzzing

The workflow of greybox fuzzing is shown in Fig. 1. The *seed evaluator* in the GBF first tries to select a seed from the queue and the selected seed is called *prioritized seed*. Then the *seed evaluator* calculates a *power schedule* for the *prioritized seed* to determine the number of new test inputs to be generated from that seed (*energy*). The *prioritized seed*, together with its *power schedule* are then passed to the *seed mutator* to mutate and generate new test inputs. Then, the *executor* executes the PUT with the mutated input. After execution, the *feedback collector* collects the runtime information such as edge coverage and helps to decide whether the test input should be kept as a new seed or not. Alternatively, if the test input causes the PUT to crash, then it is kept as a *proof-of-crash* (PoC).

[2]the CVE applications for these bugs are in progress

2.2 Problem Statement

Seed Prioritization. This problem of seed prioritization can be defined as follows:

DEFINITION 1 (SEED PRIORITIZATION). *Given a set of seeds $\mathbb{S}$, seed prioritization is to select a set of* prioritized seeds *as* S $\subseteq \mathbb{S}$ *which is a solution of the optimization problem:*

$$\begin{array}{ll} \text{Min} & \mathcal{F}(S) \\ \text{s.t.} & \mathcal{F}(S) = \sum_{s \in S} O(M_1, \cdots, M_k) \end{array} \tag{1}$$

where $O(M_1, \cdots, M_k)$ is an objective function (in practice this usually denotes a *cost*), each M_i $(i = 1, \cdots, k)$ denotes a metric that measures the quality of a seed.

Existing fuzzing tools, such as AFL, typically apply a scalarized objective function in the form of $O(M_1, \cdots, M_k) = \sum_{i=1}^{k} M_i(s)$, and M_i may measure the execution time, file size, number of covered edges etc.

Power Scheduling. Given a set of seeds $\mathbb{S}$ to be fuzzed, the power scheduling problem is to maximize the number of unique bugs discovered in a fuzz campaign that runs for a duration of fixed number of epochs [12]. Usually two types of epochs are used: fixed-run and fixed-time. This problem can be defined as follows:

DEFINITION 2 (POWER SCHEDULE). *Given a set of seeds $\mathbb{S}$ to be fuzzed and a duration of n epochs, a* power schedule *for $\mathbb{S}$ is to assign an epoch number x_s for each input $s \in \mathbb{S}$:*

$$\begin{array}{ll} \text{Max} & \mathcal{B}(\mathbb{S}) = \sum_{s \in \mathbb{S}} (\mathcal{B}(s, x_s)) \\ \text{s.t.} & \sum_{s \in \mathbb{S}} (x_s) = n \end{array} \tag{2}$$

where $\mathcal{B}(s, x_s)$ denotes the number of bugs found by exercising seed s for x_s epochs.

For AFL, its power scheduling is based on the *performance score* of the seed, which is calculated mainly based on the edge coverage and execution time of that seed. In contrast, AFLFAST allocates more *energy* to seeds that can cover rare edges (low-frequency paths in [10]).

3 MOTIVATION EXAMPLE & SYSTEM OVERVIEW

In this section, we first introduce a motivation example, and then present an overview of the proposed approach.

3.1 Motivation Example

```
1  int mjs_ffi_call(...) {// Func A
2    ...
3    mjs_parse_ffi_signature(...); // Func B
4    ...
5  }
6  int mjs_parse_ffi_signature(...) {
7    ...
8    if(*tmp_e!='(') {
9      if (mjs->dlsym==NULL) {
10       mjs_prepend_errorf(); // Func C
11       goto clean;
12     }
13     mjs->dlsym(...); // Func D
14     ...
15   }
16   else {
17     goto clean;
18   }
19   ...
20   clean:
21   ...
22 }
```

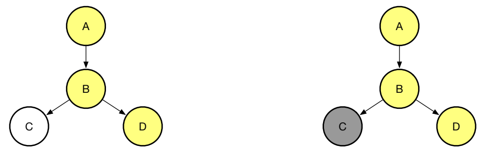
Listing 1: code snippets from *mjs*

Listing 1 is a code segment taken from *mjs* [13]. We omit the exact details and assign a short notation for each of the functions for convenience. Inside Function B, it may call C or D based on two branch conditions. Suppose a seed s_a executes the false branch at line 8 and covers functions $\{A, B, D\}$ with the function level execution trace $A \longrightarrow B \longrightarrow D$. Fig. 2 shows two different coverage states corresponding to Listing 1. At state 1 function C is not covered by any exercised seeds while at state 2 it is covered by one or more seeds. Since at state 1, s_a has the potential to be *mutated* to execute the true branch at line 8, it is reasonable to generate more new test inputs with s_a. When it is at state 2, where the true branch at line 8 has already been covered, the possibility of gaining more coverage via mutating s_a decreases. Therefore, the fuzzer should assign less *energy* for s_a at state 2. Existing GBFs are not aware of the change from state 1 to state 2. In fact, their evaluation of the quality of s_a remains unchanged between these states.

This example also shows the rationale that Cerebro uses code complexity rather than path rarity (frequency for the seeds to execute a particular path). For example, suppose input s_b goes deep into the *if* branch (high-frequency) at line 8 and s_c goes into the *else* branch (low-frequency) at line 16. If the GBF emphasizes on rare edges, like AFLFast, it will allocate more energy to s_c than s_b. However, fuzzing s_c is less beneficial because s_c will go to the cleaning logic and exit from B, missing the complex logic including function calls to C and D. Hence, when evaluating a seed, the GBF should be aware of the complexity of its execution trace as well as the complexity of the uncovered code close to the trace, which affects the *potential benefits* brought by mutating that seed.

3.2 Approach Overview

Fig. 3 depicts the basic work flow and main components of Cerebro. The input of the overall system is a set of initial seeds and the outputs are the decisions for seed prioritization and the power scheduling. Cerebro consists of four main components (the green rounded rectangles in Fig. 3): *static analyzer*, *dynamic scorer*, *multi-objective sorter* and *power scheduler*. We briefly introduce each component as follows.



(a) state 1 **(b) state 2**

Figure 2: Seed execution traces under different coverage states.*

*Legend: yellow circles denote the functions covered by seed s_a; at state 1, C (white) is not covered previously; at state 2, C (grey) has been covered by other seed(s).

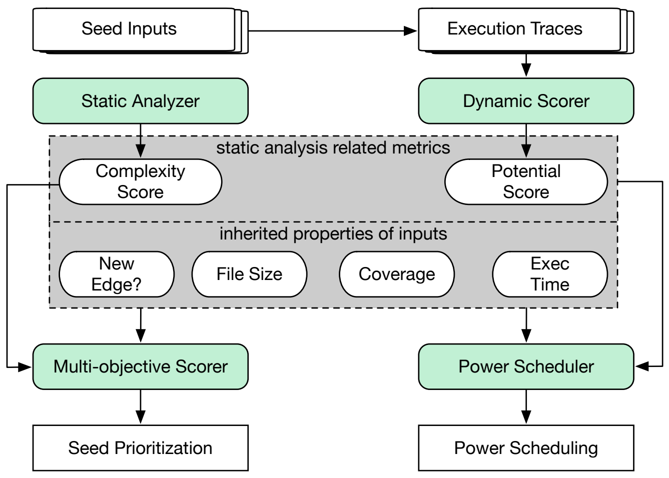


Figure 3: Overview of Cerebro *

*Legend: green rounded rectangles denote the major components of the system; squashed rectangles in the grey boxes denote scores or metrics associated with a seed.

In Cerebro, *static analyzer* scans throughout the source code and calculates a *complexity score* for each function. As every seed is associated with an execution trace, Cerebro calculates the complexity score for the trace by accumulating the complexity scores of functions on it. Details of *static analyzer* are in §4.1. The *dynamic scorer* updates the *potential scores* for the seeds on the fly. The initial values of the potential scores are derived from the complexity scores, which is elaborated in §4.2.

The outputs of *static analyzer* and *dynamic scorer* are then supplied to *multi-objective sorter* and *power scheduler* to help with the decisions of seed prioritization and power scheduling. For the seed prioritization problem defined in Def. 1, the *multi-objective sorter* utilizes a cost-effective nondominated sorting algorithm to dynamically select the prioritized set of seeds. A detailed explanation of the chosen objectives, the MOO model and the sorting algorithm is illustrated in §4.3. For the power scheduling problem defined in Def. 2, the *power scheduler* determines the energy for each seed with the outputs from both *static analyzer* and *dynamic scorer*. The *power scheduler* is elaborated in §4.4.

4 METHODOLOGY

In this section, we explain the key components in Fig. 3.

4.1 Static Analyzer

In Cerebro, static analysis is used to evaluate the complexity of each function of the PUT and lay the foundation for *dynamic scorer*. We use a mixture of two complexity metrics to evaluate the function level code complexity, namely *McCabe's Cyclomatic Complexity* (*CC*) [14] and *Halstead Complexity Measures* ($H.B$) [15]. *CC* represents the structural complexity of the function and $H.B$ represents the complexity based on the number of operations and operators. The reasons of choosing them are as follows: 1) *CC* is a widely-used complexity metric based on the number of edges and nodes in a graph; 2) $H.B$ is proven to be the second best complexity metric to indicate code vulnerability in [11]; 3) *CC* and $H.B$ complement each other, since *CC* evaluates structural complexity while $H.B$ evaluates operational complexity.

After calculating *CC* and $H.B$ for each function[3], *feature scaling* is applied to normalize each metric score separately. The normalized scores are then combined together to form the static complexity score with the following equation. This complexity score remains *unchanged* throughout the fuzzing process.

$$c\_score = \frac{norm(CC) + norm(H.B)}{2} \cdot 100 \tag{3}$$

Based on the complexity scores, we initialize the potential scores for each function. The key rationale behind the potential score is that each function brings bonus scores to the potential scores of its predecessors, and once it is covered by the fuzzer, the bonus scores are removed. Therefore, the initial potential score for a function is calculated by accumulating the bonuses brought by its successors. The formula to calculate the bonus that a function A brings to its predecessor B is:

$$bonus(A, B) = \left\lfloor \frac{c\_score_A}{2^{Distance(B,A)}} \right\rfloor \tag{4}$$

where $\lfloor \cdot \rfloor$ denotes the floor function, c_score_A is the complexity score of A and $Distance(B, A)$ is the shortest distance from B to A in call graph.

For example, in Fig. 4a, A, B, C, D, E, F and G denote functions in the program. Assume F has a complexity score of 9. According to Equation 4 the bonuses it brings to its predecessors are: 4 for D ($\lfloor \frac{9}{2^1} \rfloor$), 2 for B ($\lfloor \frac{9}{2^2} \rfloor$) and 1 for A ($\lfloor \frac{9}{2^3} \rfloor$). Similarly, the bonus G brings to D is 2. Thus, because D has F and G as its successors, its dynamic score is initialized to be 6 (4 + 2).

4.2 Dynamic Scorer

The purpose of the dynamic scorer is to evaluate the complexity of the code that is not covered by the fuzzer presently but could possibly get covered through mutations. Although a thorough examination of coverage with external tools like *gcov* [16] can achieve this, it will greatly decrease the execution speed of the PUT. Furthermore, calculating accurate coverage on basic-block level can introduce significant overhead to GBFs. Due to these reasons, we propose the following approach for function-level *dynamic potential evaluation* (DPE).

[3]Due to page limit, we omit the formula to calculate CC and $H.B$. Interested readers can refer to [11] for details.

Given a function A, the calculation of potential score is:

$$p\_score(A) = \sum_{F \in S_A} bonus(F, A) \tag{5}$$

where $bonus(F, A)$ is the bonus score that $F \in S_A$ brings to A, and S_A is the set of uncovered successors of A.

After the dynamic potential score is calculated, it is combined with the static complexity score. The combined score will be supplied to the power scheduler as a parameter (§ 4.4). Given a function A, the combined score is:

$$combined\_score(A) = c\_score(A) + p\_score(A) \tag{6}$$

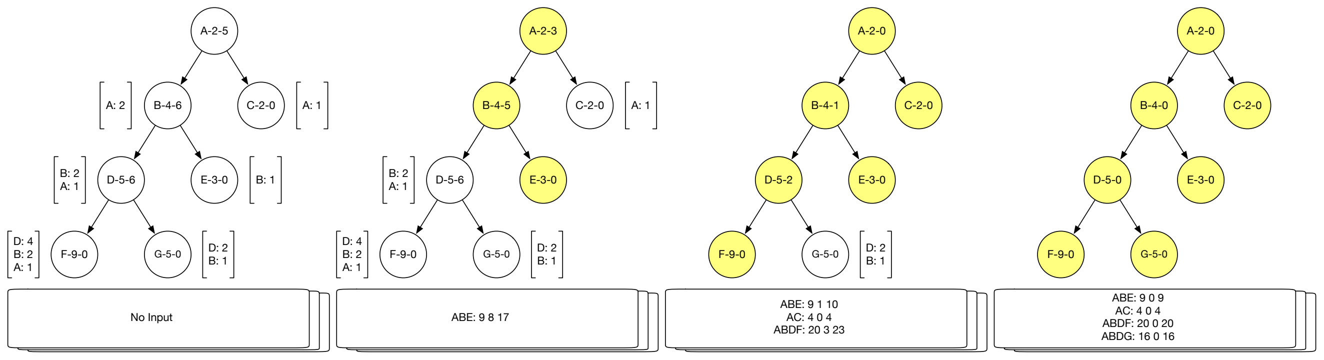
where $c\_score(A)$ is the static complexity score for A and $p\_score(A)$ is the dynamic potential score for A.

Here, we use a step-by-step example shown in Fig. 4 to illustrate how Cerebro performs DPE. Each subfigure shows a coverage state held by the fuzzer. Each node in Fig. 4 is a function — yellow nodes are covered functions and white nodes are uncovered functions. Fig. 4a shows the state before fuzzing. After given an initial input covering function A, B and E (named as *ABE* for convenience), the fuzzer holds the state shown in Fig. 4b. Since B is covered, the bonus it brings to A is removed — the function-value pair $A : 2$ for node B is removed from Fig. 4b. So the potential score of A is updated to 3 (5 − 2). The potential score of B is updated to 5 since its successor E is covered. Thus, the potential score of input *ABE* is calculated by accumulating the potential scores of each function on its execution trace — the potential score of input *ABE* is 8 (3 + 5 + 0). Similarly, the complexity score of input *ABE* is calculated by accumulating the complexity scores of each function on its execution trace — the complexity score for *ABE* is 9 (2 + 4 + 3), and it never changes through out the fuzzing process. Thus, the final combined score of *ABE* is 17 (9 + 8) in Fig. 4b. After generating two more inputs *AC* and *ABDF*, the fuzzer holds the state shown in Fig. 4c. Then, the static complexity score for input *ABE* remains 9 while its potential score drops to 1 (0 + 1 + 0). The combined score of *ABE* now is 10 (9 + 1). We can clearly see that as more functions under the trace of *ABE* are covered, the potential score and the total score of *ABE* decrease. Note that fuzzers without awareness of input potentials do not distinguish between state 2 and state 3 and thus will not adjust power scheduling accordingly. Finally, assuming the fuzzer covers the last function G, now it holds the state shown in Fig. 4d. Since all functions are covered, all input potentials are used up (drop to 0) and the combined score of a function is now only determined by its complexity score.

4.3 Multi-objective Sorter

The purpose of the multi-objective sorter (*MO sorter*) is to prioritize the seeds via various metrics. In this section, we first introduce the objectives for prioritizing seeds, then describe the MOO model addressing the problem defined in Def. 1, and finally present the nondominated sorting algorithm which solves the problem based on the MOO model.

4.3.1 Metrics. The metrics for seed prioritization are *file size* (M_1), *execution time* (M_2), *number of covered edges* (M_3), *whether the seed*



(a) step 1 **(b) step 2** **(c) step 3** **(d) step 4**

Figure 4: A step-by-step demonstration of the dynamic scoring algorithm*

*Legend: Each node is a function. For example, in F-9-0, the letter (F) is the function name, the first number (9) in the node is the complexity score, the second number (0) is the potential score. The function-value pairs (e.g., D:4 for node F in Fig.4a) associated with the curly brackets are the bonus scores that the function brings to its predecessors.

brings new edge coverage (M_4) and its *static complexity score of the execution trace* (M_5)[4]. The rationale of using these metrics is:

- *file size*: smaller seeds are more compact — mutations on them are more likely to hit interesting bytes.
- *execution time*: seeds with shorter execution time can increase the average speed of the PUT and consequently the fuzzing efficiency.
- *number of covered edges*: seeds with higher coverage are preferred, like in most GBFs — intuitively, fuzzing a seed with good coverage might generate more test inputs with good coverage.
- *whether the seed brings new edge coverage*: seeds covering new edges are preferred because fuzzing such seeds are more likely to bring new coverage.
- *static complexity score of the execution trace*: seeds covering more complex code are preferred because intuitively complex code tends to be more error-prone [11].

Our model to handling the metrics takes into account two facts. 1). Some of these metrics are conflicting each other, e.g., a seed covering more complex code may execute slower. 2). The scalarization model adopted by existing GBFs needs a proper weighting schema, which may vary from project to project. Thus, we need a MOO model to balance between these metrics.

4.3.2 MOO Model for Seed Prioritization.

With the above multiple metrics, a new MOO model can be inferred as follows:

Definition 3 (Multi-objective Seed Prioritization). *Given a set of seeds $\mathbb{S}$, multi-objective seed prioritization is to select a set of seeds* S*:*

$$\text{Min}\big(\vec{\mathcal{F}}(\mathbb{S})\big) = \text{Min}\big(O_1(s), O_2(s)...O_k(s)\big), s \in \mathbb{S} \tag{7}$$

where $\vec{\mathcal{F}}(\mathbb{S})$ is an objective vector that denotes k objective functions ranging from O_1 to O_k.

In our case, $k = 5$ as we have 5 metrics. Instead of solving the problem in Def. 1 by scalarizing different metrics into a single objective function, we set a separate objective for each metric in our MOO model. Here is the mapping between metrics and objectives: let $M_k(s)$ denote the k-th metric for seed $s \in \mathbb{S}$, we have $\text{Min}(O_1(s)) = \text{Min}(M_1(s)), \text{Min}(O_2(s)) = \text{Min}(M_2(s)), \text{Min}(O_3(s)) = -\text{Max}(M_3(s)), \text{Min}(O_4(s) = -\text{Max}(M_4(s)), \text{Min}(O_5(s)) = -\text{Max}(M_5(s))$. The reason behind the mapping is that, we want to minimize M_1, M_2 but maximize $M3$, M_4, M_5.

4.3.3 Pareto frontier.

Pareto frontier calculation is one of the most commonly used posteriori preference techniques for MOO problems. In our proposed MOO model, the prioritized seeds are inside the Pareto frontier of the entire set of seeds. Given a set of the seeds $\mathbb{S}$ and an objective vector $\vec{\mathcal{F}} = [f_1, f_2, \cdots, f_k]$, we say s dominates($\prec$) s' *iff*:

$$f_i(s) < f_i(s'), \quad \forall i \in \{1, 2, \cdots, k\}$$

where $s, s' \in \mathbb{S}$; the Pareto frontier(P) is defined as [17]:

$$P(\mathbb{S}) = \{s \in \mathbb{S} \mid \{s' \in \mathbb{S} \mid s' \prec s, s' \neq s\} = \emptyset\} \tag{8}$$

Theoretically, the calculation of Pareto frontier requires to compare each seed against all the other seeds to check their domination relation. However, as shown in Fig. 1, $\mathbb{S}$ keeps expanding as new seeds come in. It is costly to recalculate the Pareto frontier every time a new seed is added. To tackle this problem, we propose Algo. 1 based on the *nondominated sorting* in [18].

4.3.4 Nondominated Sorting.

In greybox fuzzing, the seeds are stored in a queue as shown in Fig. 1. To efficiently calculate the Pareto frontier, the entire queue is splitted into four sub-queues as shown in Fig. 5. They are *Frontier* (P), *Dominion* (D), *Recycled* (R) and *Newly Added* (N).

The basic idea is to keep popping seeds from P for fuzzing, which requires P to be maintained as the Pareto frontier of the seeds not yet fuzzed in the current cycle. To efficiently calculate the Pareto frontier, we adopt the concept of *rank* from nondominated sorting. Each seed is associated with a rank representing the domination relation. Intuitively, the value of the rank for a seed represents the number of seeds dominating it. For instance, seeds with rank 0 are not dominated by any other seeds so they are on the Pareto frontier; seeds with rank 1 are only dominated by seeds with rank 0; seeds with rank 2 are only dominated by seeds with rank 0 and 1, and so

[4]Potential score is not used in seed prioritization (see Fig. 3). In implementation, only when the fuzzer chooses to fuzz a seed, its potential score will be *lazily updated* — the actual potential score is calculated *on the fly* after a seed is selected to be fuzzed, since it is too costly to update the potential score for every seed every time whenever a new function is covered.

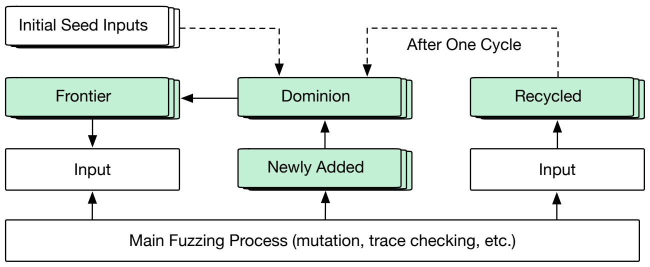


Figure 5: Structure of the seed queue in Cerebro

on. So given a set of seeds $\mathbb{S}$ and their ranks $\mathbb{R}$, P can be calculated as:

$$P(\mathbb{S}) = \{s \in \mathbb{S} \mid s.rank = min(\mathbb{R})\} \tag{9}$$

In Algo. 1, the variable `queue_rank` maintains the $min(\mathbb{R})$ in Equation 9. R is used to store the seeds fuzzed in the current cycle. D is to store the seeds that are not fuzzed in the current cycle and are dominated by the seeds in P. N is to temporarily store the interesting seeds kept by the fuzzer through fuzzing the seed popped from P. The logic for updating ranks according to domination relation between seeds corresponds to line 5 – line 10. The calculation of P corresponds to line 29 – line 32. Each time the fuzzer uses up the seeds in P, a new Pareto frontier will be calculated based on N and D and stored into P (line 12 to line 32 in Algo. 1). If both N and D are empty during calculation of the new P, it indicates that the fuzzer has fuzzed every seed in current cycle and a new cycle starts. Now that every seed is stored in R, the fuzzer will move the seeds from R to D and calculate P accordingly (line 19 to line 32).

The benefit of maintaining the *rank* is to avoid redundant domination relation checks between seeds that are already compared with each other. Assuming there are n seeds in N and d seeds in D, $(d + n) \cdot (d + n - 1)$ checks are needed without maintaining the ranks. However, only $n \cdot (d + n - 1)$ checks are needed with the ranks maintained, saving a total number of $d^2 + nd - 1$ checks.

4.4 Power Scheduler

The purpose of power scheduler is to assign proper energy to the seeds selected by the *MO sorter* addressing the problem defined in Def. 2. Most GBFs calculate the energy for a seed as follows[5]:

$$energy = allocate\_energy(q_i) \tag{10}$$

where q_i is the quality of the seed.

In Cerebro, the *power scheduler* incorporates the combined score of the static complexity and the dynamic potential for seed energy allocation:

$$energy' = energy \cdot s_f \cdot \frac{i_s}{a_s} \tag{11}$$

where *energy* is the energy calculated in Equation 10, s_f is the score factor, i_s is the combined score of the seed and a_s is the average combined score of all the seeds.

The combined score enables the fuzzer to fully exploit the knowledge about the complexity of both covered and uncovered code.

[5]For clarity, we omitted some details in Eqt. 10 and Eqt. 11. In actual implementation for both Cerebro and *AFL*, the fuzzers utilize several different metrics to calculate q_i (a.k.a. *performance_score* in [5] or $\alpha(i)$ in [10]).

Algorithm 1: Cycle-based Nondominated Sorting

```
1  $P = D = N = R = \emptyset$;
2  load $D$ with the user provided test seed(s);
3  queue_cycle = 0;
4  queue_rank = -1;
5  def update_ranks(a: seed, b: seed):
6      if $a \neq b$ then
7          if $a \prec b$ then
8              $b$.rank += 1
9          if $b \prec a$ then
10             $a$.rank += 1;
11 while not (time budget reached or kill signal received) do
12     if $P$ is $\emptyset$ then
13         $n$.rank = queue_rank + 1 for n in N;
14         $D = D \cup N$;
15         for $n$ in $N$ do
16             for $d$ in $D$ do
17                 update_ranks($d$, $n$);
18         $N = \emptyset$;
19         if $D$ is $\emptyset$ then
20             queue_cycle += 1;
21             queue_rank = -1;
22             r.rank = 0 for r in R;
23             for $r$ in $R$ do
24                 for $r'$ in $R$ do
25                     update_ranks($r$, $r'$);
26             $D = D \cup R$;
27             $R = \emptyset$;
28         queue_rank += 1;
29         for $d$ in $D$ do
30             if $d$.rank == queue_rank then
31                 pop $d$ from $D$;
32                 push $d$ into $P$;
33     pop $s$ from $P$;
34     update power schedule for $s$;
35     fuzz $s$;
36     push $s$ into $R$;
```

The static complexity score indicates the inherent complexity of the execution trace of a seed, while the dynamic potential score indicates the complexity of uncover code near the execution trace. By combining both scores, Cerebro allocates more energy for seeds with better potential of leading to bugs or new edges.

5 IMPLEMENTATION AND EVALUATION

We implement a novel fuzzer, Cerebro. A more detailed introduction about Cerebro, together with the raw experiment data and a demonstration kit, is available at http://sites.google.com/site/cerebrofuzzer/. Specifically, the static complexity analyzer is written based on Clang's [19] Python binding and *lizard* [20] with about 800 lines of Python. The instrumentation module used to track the execution traces is implemented in 1400 lines of C++ on top of LLVM [21] framework. The core dynamic fuzzing logic of Cerebro is written with around 10,000 lines of Rust Code.

5.1 Evaluation Setup

Evaluation Dataset. The evaluation is conducted on 8 widely-used real-world open source programs, each of which is from a different project in a different domain.

Program	size (B)	Stars	Duration	Project
fuzzershell	3.7M	851	18y	SQLite [22]
mjs	291K	686	2y8m	MJS [13]
nm	6.0M	-	28y	Binutils [23]
cxxfilt	13M	-	28y	Binutils [23]
pngfix	313K	268	23y2m	libpng [24]
radare2	60M	8002	13y	Radare2 [25]
xed	16M	732	2y8m	Intel XED [26]

Table 1: Details of the evaluated programs

mjs is a light weight javascript engine for embedded system [13]. *pngfix* is a tool from *libpng* [24]. *fuzzershell* is the official fuzzing harness of *sqlite* [22]. *xed* is the disassembler used in *Intel PIN* [27]. *radare2* is a famous open source reverse engineering tool [25]. *cxxfilt* is a program in *GNU Binutils* [23] to demangle C++ function names. *nm* is a program from *GNU Binutils* to list symbols in an object file. Specifically, we used the same version (version 2.15) of *cxxfilt* and *nm* as in the AFLFAST paper for more direct comparison. Furthermore, we also ran CEREBRO on a newer version (version 2.29) of *nm* in hope of finding new bugs. For clarity, we denote *nm* version 2.15 as *nm(old)* and *nm* version 2.29 as *nm(new)*.

From Table 1, we can see that the selected programs are very popular in the community and their sizes vary from a few hundred KBs to tens of MBs. We can also see that the projects are long-time supported. The diverse dataset helps to demonstrate the generality and scalability of CEREBRO.

Evaluated Tools. We compared CEREBRO with two other fuzzers, namely AFL and AFLFAST [28]. AFL is by far the most popular GBF. AFLFAST represents the state-of-the-art seed prioritization and power scheduling techniques. All the mutation operations are the same across the tools and only non-deterministic mutation operations are used because power scheduling does not affect the deterministic mutations [29].

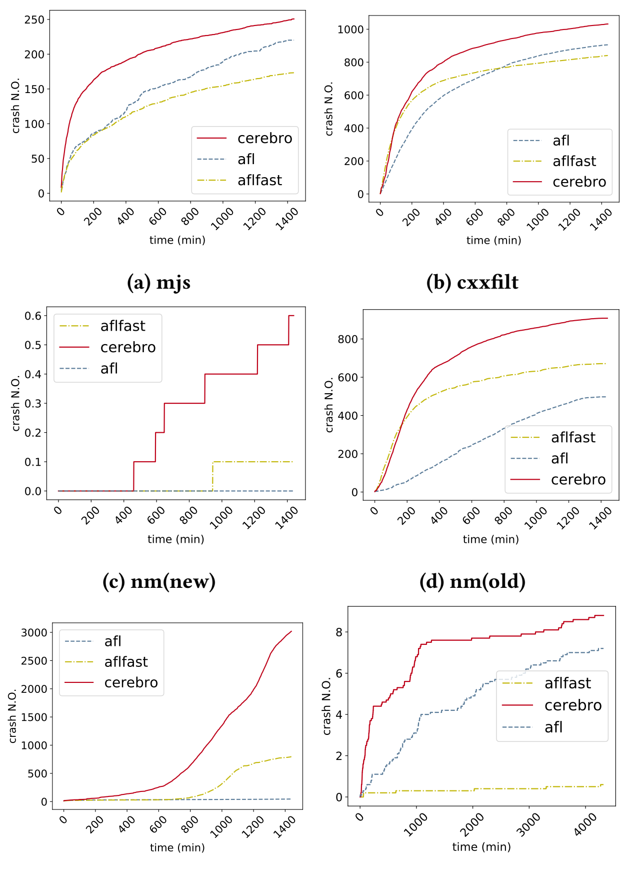
Experimental Infrastructure. We conducted all our experiments on a machine of 12 Intel(R) Xeon(R) CPU E5-1650 v3 cores and 16 GB memory, running a 64-bit Ubuntu 16.04 LTS system. Each experiment was repeated for 10 times and each took 24 hours except for *radare2*, which took 72 hours due to its slow execution speed.

Research Questions. We aim to answer these questions:
RQ1. How is the crash detection capability of CEREBRO?
RQ2. How is the vulnerability detection capability of CEREBRO?
RQ3. How do seed prioritization and power scheduling in CEREBRO affect the performance separately?

5.2 Crashes (RQ1)

During the experiments, we found crashes in six programs: *mjs*, *cxxfilt*, *nm(old)*, *nm(new)*, *xed* and *radare2*. We follow AFL's definition of unique crash, i.e., two crashing inputs with the same edge trace are counted as the same crash. Unique crash serves as a good indicator of a fuzzer's capability of exercising error-prone paths [10, 30].

Fig. 6 shows the average number of *unique crashes* found over time in 24 hours by CEREBRO, AFLFAST, and AFL for ten runs. In general, it is obvious that CEREBRO significantly outperforms AFL



(a) mjs (b) cxxfilt
(c) nm(new) (d) nm(old)
(e) xed (f) radare2

Figure 6: Unique crashes found over time (Higher is better)

and AFLFAST as to unique crashes found over time. From the trends of the plots, we can see that CEREBRO not only finds more crashes after 24 hours but also finds them faster. This indicates CEREBRO is both efficient and effective for crash exposure.

In particular, we can see that all the fuzzers found less crashes in the newer version of *nm* and AFL even failed to find any crash in 24 hours. This is as expected because the developers patched the bugs reported in the old version. On one hand, for the old version with much more bugs, CEREBRO can find considerably more crashes (at least 30%) than both AFL and AFLFAST. On the other hand, for the new version with existing bugs fixed and new bugs introduced, CEREBRO can still find substantially more crashes with a shorter time.

Table 2 shows the statistic test results for this experiment. As suggested by Klees et al. [31], we adopted the Mann Whitney U-test and the Vargha-Delaney $\hat{A}_{12}$ value [32] here. The Mann Whitney U-test measures the statistical significance of the results. The Vargha-Delaney $\hat{A}_{12}$ value measures if we randomly pick one run out of the ten runs for both CEREBRO and its competitor, the probability that CEREBRO performs better.

In general, we can see that CEREBRO finds significantly more crashes than both AFL and AFLFAST. Most of the $\hat{A}_{12}$ values are also above the conventionally large effect size (0.71) [33]. The pitfall comes to the case of *nm(new)*, where all the fuzzers tends to find

Project	Average Crash #			$\hat{A}_{12}$	
	C	AF	A	AF	A
mjs	250.7	173.3	223.8	**0.97**	**0.85**
cxxfilt	9	3	5	**1.0**	**0.92**
nm(old)	10	5	10	**0.91**	**1.0**
nm(new)	0.6	0.1	0	0.56	0.6
radare2	8.8	0.6	7.2	**1.0**	0.655
xed	3019.0	797.0	44.7	**0.9**	**1.0**

Table 2: Details of the unique crashes detected*

*statistically significant results by Mann-Whitney U Test are marked as bold; C stands for CEREBRO; AF stands for AFLFAST; A stands for AFL.

very few crashes. Despite that, CEREBRO still performs a bit better than the other two tools on *nm(new)*. We can conclude that CEREBRO performs better than AFL and AFLFAST in terms of crash detection from the statistic point of view.

From the analysis of Figure 6 and Talbe 2, we can positively answer **RQ1** that CEREBRO *significantly* outperforms the state-of-the-art fuzzers in terms of crash detection.

5.3 Vulnerabilities (RQ2)

As suggested by Klees et al., besides unique crashes, researchers should also use unique bugs as the ground truth for fuzzer evaluation [31]. This is because several unique crashes could be related to one unique bug with the same root cause. After scrutiny, we further classified all the unique crashes from these programs into 16 unique previously unknown bugs according to how the developers apply patches[6]. Notably, the bug in *nm(new)* is later verified to be a reproducer of CVE-2017-13710.

Figure 7 is a boxplot showing the Time-To-Exposure (TTE) for the bugs that every fuzzer can find for at least four out of ten experiment runs. The y-axis is time. The dark bars inside the boxes are the median values. The top border of a box indicates the 75th percentile and the bottom border of a box indicates the 25th percentile. Hence, a lower position of the box indicates a shorter TTE for a bug, which means better performance. Moreover, a small box size means the variance between every run is small and the fuzzer's performance is stabler.

In general, we can see that the CEREBRO can detect the bugs with a shorter time comparing to AFLFAST and AFL. On average, CEREBRO can find the bugs 4.7 times and 2.9 times faster than AFLFAST and AFL respectively. In most cases, the box of CEREBRO is also smaller than AFLFAST's and AFL's, which means that CEREBRO can stably detect those bugs earlier than them. However, in some cases, the box size of AFLFAST is the smallest. This is because AFLFAST finds the bug in fewer runs and the performance appears to be stabler. For example, in *xed invalid-read-3*, AFLFAST only finds the bug in four experiment runs while CEREBRO finds the bug in all the ten runs (see Table 3) and one or two outliers make the box of CEREBRO larger in size.

Table 3 shows the statistic test results for this experiment. The *# runs* column shows the number of runs that a fuzzer can detect a

[6]We filtered out assertion fails, which also contribute to the crashes found. The assertion fails have also been reported and fixed.

Project	Bug/CVE ID	# runs			$\hat{A}_{12}$	
		C	AF	A	AF	A
mjs	BufferOverflow	9	4	9	**0.77**	**0.78**
	UseAfterFree-1	3	-	1	-	**0.69**
	UseAfterFree-2	10	10	9	0.53	**0.96**
	InvalidRead	10	9	10	**0.65**	0.68
	NegSizeParam	10	8	9	0.66	0.72
	StackOverflow	3	-	-	-	-
	FloatPointError	5	1	3	**0.72**	0.66
xed	InvalidRead-1	10	10	9	**0.72**	**0.73**
	InvalidRead-2	4	1	4	**0.67**	0.52
	InvalidRead-3	10	4	6	**0.94**	**0.81**
	InvalidRead-4	10	10	10	0.65	0.89
	InvalidRead-5	7	2	1	**0.82**	**0.84**
	InvalidRead-6	10	4	7	**0.97**	**0.77**
	InvalidRead-7	9	3	5	**0.88**	**0.81**
radare2	2018-xxxx	10	5	10	**0.98**	**0.88**
nm	2017-13710	2	1	-	0.56	-
mean	-	7.63	4.50	5.81	0.78	0.80

Table 3: Details of the unique bugs discovered*

*"-" means no bug is found in 10 runs; statistically significant results by Mann-Whitney U Test are marked as bold; C stands for CEREBRO; AF stands for AFLFAST; A stands for AFL.

particular bug in ten runs. The $\hat{A}_{12}$ column shows the $\hat{A}_{12}$ values calculated based on the TTEs and the value is marked as bold if the U-test shows significance.

In general, we can observe from Table 3 that CEREBRO can find 9 of the 14 bugs significantly faster than AFL (the 2 bugs that AFL cannot detect are not counted);.and 10 of the 14 bugs significantly faster than AFLFAST (the 2 bugs that AFLFAST cannot detect are not counted). We can also see that the mean $\hat{A}_{12}$ values are 0.78 against AFLFAST and 0.8 against AFL, both of which are above the conventionally large effect size (0.71) [33]. The results suggest that CEREBRO can outperform both AFLFAST and AFL from the statistical aspect.

Case Study. To demonstrate the reason behind CEREBRO's superiority, we present the case of CVE-2018-xxxx, where CEREBRO has a high confidence of being better ($\hat{A}_{12}$ is 0.98 against AFLFAST and 0.88 against AFL). This CVE is triggered by an invalid memory read caused by a `strlen` function call where the pointer passed to `strlen` points to zero page due to a missing length check for the parent string containing the pointer. The patched function is `r_bin_dwarf_parse_comp_unit`, as the function code shown in Listing 2. The true branch of the *if* condition at line 3 is a relatively rare branch and AFLFAST will prioritize the seeds exercising this branch. However, this branch could mistakenly guide the fuzzer away from triggering the bug. This explains the reason why AFLFAST found the bug only in 5 out of 10 runs while CEREBRO and AFL can detect it in all 10 runs. Comparing to AFL, CEREBRO allocates more energy to the seeds reaching `r_bin_dwarf_parse_comp_unit` meanwhile not getting into the true branch of the *if* condition on line 7. The reason is that function `sdb_set` keeps contributing bonus potential scores to its predecessor `r_bin_dwarf_parse_comp_unit` until it is covered and the bug is triggered. Since CEREBRO manages to generate more new inputs from the seeds close to but not yet reaching the buggy function, it can usually detect the bug faster than AFL.

```
ut8* r_bin_dwarf_parse_comp_unit(...) {
  ...
  if(cu->hdr.length > debug_str_len)
  return NULL;
  while (...) {
  ...
    if(...){
    // missing check for the length of "name"
       sdb_set(s, "DW_AT_comp_dir", name, 0);
    }// end of if
  ...} // end of while
}
```

Listing 2: code snippets from *radare2*

From the analysis of Figure 7, Table 3 and the case study, we can positively answer **RQ2** that CEREBRO *significantly* outperforms the state-of-the-art fuzzers in terms of vulnerability detection.

5.4 Evaluation of Individual Strategies (RQ3)

CEREBRO uses an MOO-based seed prioritization strategy together with an input potential aided power scheduling strategy. To analyze the effects of each individual strategy, we configure 3 variants of CEREBRO: *Cerebro-base* uses no seed prioritization and allocates a constant energy to every seed. *Cerebro-moo-only* uses only the MOO-based seed strategy and allocates a constant energy to every seed. *Cerebro-pot-only* uses no seed prioritization and applies the input potential aided power scheduling. We evaluated these strategies individually with two programs — *mjs*, representing relatively small programs and *xed*, representing relatively large programs (see Table 1). Fig 8 shows the number of crashes detected over time for each individual strategy. In general, both strategies can help to detect more crashes within the time budget.

In particular, on *xed*, the MOO-based seed prioritization strategy seems to restrain the crash detection performance as it needs to balance between exploitation and exploration as discussed earlier. However, the seed prioritization strategy does help to boost crash detection as the queue of seeds grows larger and it surpasses the power scheduling strategy after around 1000 minutes. This explains why *Cerebro-pot-only* has a better performance than CEREBRO in the first 12 hours but gets surpassed later.

If we compare the results of *mjs* and *xed*, we can see that the performance improvement over the baseline strategy (*Cerebro-base*) on *xed* is much more significant than on *mjs*. The rationale is that as *xed* is a much larger program than *mjs* (see Table 1), the fuzzers need to keep much more seeds in queue for *xed* and to evaluate the quality of seeds properly and allocate the computational resource precisely becomes much more important. As a result, both strategies becomes more effective on larger programs.

Lastly, combining both strategies leads to the best results for finding crashes on both projects, which implies the two strategies can mutually benefit each other without conflicts. Thus, it makes sense that CEREBRO needs to combine the seed prioritization strategy with a proper power scheduling strategy by allowing better seeds to produce more test inputs to boost the final coverage after convergence.

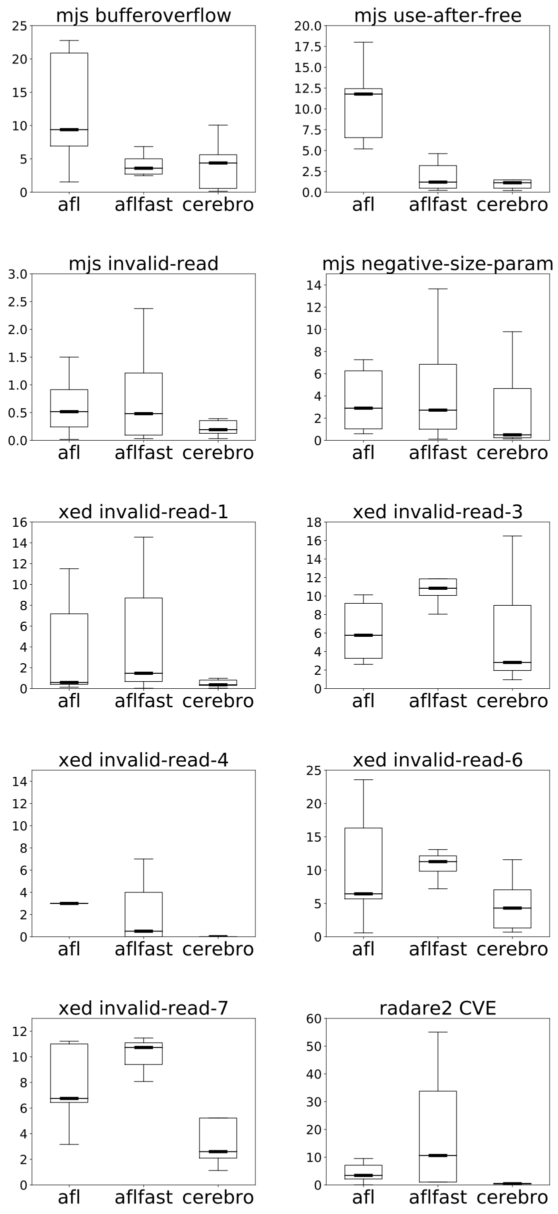
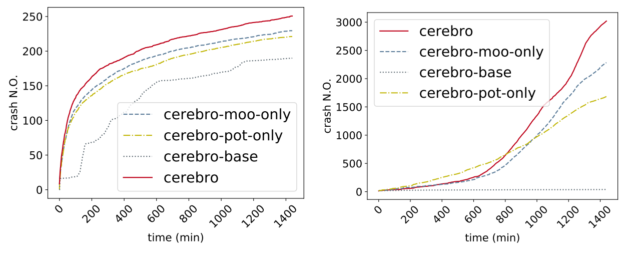


Figure 7: Time-To-Exposure for bugs (Lower is better)
*The y-axis is the time and the unit is *hour* except for *xed invalid-read-4*, which is *second*.

From the analysis of Figure 8, we can positively answer **RQ3** that the seed prioritization and the power scheduling can help to improve the fuzzing performance separately; combining them can further improve fuzzing efficiency.

5.5 Discussion

Threats to validity. The threats of validity come from three aspects. First, although complexity is generally considered as the enemy of software security [34], some researches are skeptical



(a) mjs **(b) xed**

Figure 8: Crashes found over time with different strategies (Higher is better)

about it [35]. After analyzing the bugs, our empirical finding is that typically the program executes a series of complex functions, makes a logical fault somewhere and eventually crashes in a possibly simple function. So using the execution trace-level complexity is reasonable. Second, we used only two complexity metrics (*CC* and *H.B*). As there exist a vast variety of code complexity metrics (e.g., Jilb metrics [36], CoCol metrics [36], Oviedo metrics [37], etc.), we will seek more combinations to facilitate the input potential in future. Third, results of tool evaluations could always be taken with a grain of salt. For instance, we set the time limit of most experiments to be 24 hours because the edge coverage of most programs we test tends to converge in 24 hours. In future, we will try more setups from different aspects and exhibit the results on different levels of details.

Additional Experiments. Last but not least, we conducted extra experiments to better evaluate Cerebro from various aspects. For example, to check how well Cerebro complements constraint breaking techniques, we combine Cerebro with dictionary-based mutation operators and conduct experiments to check the coverage gain. The results of combining with dictionary-based mutation show that Cerebro can better complement orthogonal techniques. We also conducted experiments with Cerebro by allowing it to export the seed queue information during execution to evaluate the quality of seeds on the Pareto frontier. The results demonstrate that the quality of seeds on the Pareto frontier is indeed better. These extra experiment results and data are available on our website [38] for interested readers.

6 RELATED WORK

Instead of listing all the related works, we focus on fuzzing and random testing techniques related to Cerebro.

Seed Prioritization. Several techniques have been proposed [6, 10, 11, 39–42] for offline seed selection and online seed prioritization. Shudrak *et al.* [11] propose an approach to evaluating and improving black-box fuzzing effectiveness by prioritizing the seeds which exercise high complexity codes. As they use software complexity as the sole goal for seed prioritization, the results heavily rely on the quality of the complexity analysis. In Cerebro, we propose a MOO model to balance between different objectives instead of only considering the execution trace complexity. In Vuzzer [6], the error handling paths are deprioritized. This intuition aligns with the concept of input potential. However, input potential is more general as error handling code can be quite complex and bug-prone sometimes [43]. Both Vuzzer and AFLFast [10] prioritize seeds exercising low-frequency paths. However, as shown in §3.1 and §5, the assumption that fuzzing such seeds brings more benefits does not always hold. In CollAFL [30], three different strategies are proposed to prioritize seeds with more neighbor branches or descendants. The intuition of bringing awareness of uncovered code to the fuzzer is similar to Cerebro, whereas Cerebro utilizes the complexity of near neighbors instead of the quantity of direct neighbors in [30]. Seed prioritization is also performed in directed fuzzing [33, 44] to help the fuzzer to reach the targets as early as possible.

Power Scheduling Techniques. AFLFast [10] assigns more energy to seeds exercising the low-frequency paths. In AFLGo [33], a simulated annealing-based power schedule is proposed to increase energy for seeds "closer" to the target locations. The purposes of directed fuzzing and Cerebro are different. The purpose of directed fuzzing is to reach the given specific targets as fast as possible. Thus, as pointed out by Serebryany [45], the coverage of directed fuzzers may not be good, since the power scheduling leans towards only the targets. The purpose of Cerebro, however, is to discover as many bugs as possible given a time budget.

In comparison with these seed prioritization and power scheduling techniques, Cerebro explicitly presents the concept of input potential and proposes a MOO model to incorporate different important and conflicting objectives.

Constraint Breaking & Fuzzing Boosting Techniques. Many other techniques in fuzzing are orthogonal to Cerebro. In greybox fuzzing, a hot topic is about penetrating through path constraints. Plenty of approaches have been proposed to address this problem. SAGE [46] and Driller [47] use symbolic/concolic execution to solve the path constraints for fuzzers. Vuzzer uses dynamic taint analysis to tackle path constraints. Steelix [7] utilizes comparison progress information to solve magic-byte comparisons. Angora [8] adopts byte-level taint analysis and a gradient-descent algorithm for constraint penetration. These techniques often involves modifications of the *seed mutator*, *executor* or *feedback collector* in Fig. 1 but not the *seed evaluator*, making them orthogonal to Cerebro. Several techniques are proposed to boost the performance of fuzzing from different aspects [12, 48–53]. Among them, FairFuzz [50] utilizes masks to preserve key patterns inside a seed to reach certain edges. FairFuzz targets rare edges with the help of the seed masks. However, the idea of input masks can be applied to target any edges, not just rare edges, which makes it orthogonal to Cerebro.

7 CONCLUSION

In this paper, we propose the concept of input potential and a MOO model for seed evaluation in greybox fuzzing. With the complexity of covered code and the potential of uncovered code based on execution traces, we address two general problems of greybox fuzzing, namely seed prioritization and power scheduling. We conduct evaluations on 8 real-world programs. Results exhibit that Cerebro performs significantly better in bug detection and program coverage than state-of-the-art techniques. In future, we plan to complement Cerebro with other orthogonal fuzzing techniques.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[2] "A survey of some free fuzzing tools," https://lwn.net/Articles/744269/, accessed: 2018-04-01.

[3] "Basic blackbox fuzzing – ansvif," https://github.com/oxagast/ansvif/wiki/Basic-Blackbox-Fuzzing, accessed: 2018-04-01.

[4] V. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *ASE*. ACM, 2016, pp. 543–553.

[5] "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, accessed: 2018-04-01.

[6] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *NDSS*, Feb. 2017. [Online]. Available: https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf

[7] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 627–637. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106295

[8] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *CoRR*, vol. abs/1803.01307, 2018. [Online]. Available: https://arxiv.org/abs/1803.01307v2

[9] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," 2018. [Online]. Available: https://nebelwelt.net/publications/files/18Oakland.pdf

[10] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1032–1043. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978428

[11] M. O. Shudrak and V. V. Zolotarev, "Improving fuzzing using software complexity metrics," in *Information Security and Cryptology - ICISC 2015 - 18th International Conference, Seoul, South Korea, November 25-27, 2015, Revised Selected Papers*, 2015, pp. 246–261. [Online]. Available: https://doi.org/10.1007/978-3-319-30840-1_16

[12] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *ACM Conference on Computer and Communications Security*. ACM, 2013, pp. 511–522.

[13] "mjs," https://github.com/cesanta/mjs, accessed: 2018-04-01.

[14] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[15] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[16] "Gnu gcov," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, accessed: 2018-04-01.

[17] H. Ishibuchi, N. Tsukamoto, and Y. Nojima, "Evolutionary many-objective optimization: A short review," in *IEEE Congress on Evolutionary Computation*. IEEE, 2008, pp. 2419–2426.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[19] Clang. Clang: a c language family frontend for llvm. [Online]. Available: https://clang.llvm.org/

[20] "Lizard." [Online]. Available: https://github.com/terryyin/lizard

[21] "Llvm," https://llvm.org/, accessed: 2018-04-01.

[22] "Sqlite: a sql database engine," https://www.sqlite.org/index.html, accessed: 2018-04-01.

[23] "Gnu binutils," https://www.gnu.org/software/binutils/, accessed: 2018-04-01.

[24] "libpng," http://www.libpng.org/pub/png/libpng.html, accessed: 2018-04-01.

[25] "radare2: reverse engineering framework," https://github.com/radare/radare2, accessed: 2018-04-01.

[26] "Intel x86 encoder decoder (intel xed)," https://github.com/intelxed/xed, accessed: 2018-04-01.

[27] "Pin - a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, accessed: 2018-04-01.

[28] "Aflfast (extends afl with power schedules)," https://github.com/mboehme/aflfast, accessed: 2018-04-01.

[29] "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: 2018-04-01.

[30] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 1–12.

[31] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[32] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[33] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2329–2344. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134020

[34] "More complex = less secure," http://www.mccabe.com/pdf/More%20Complex%20Equals%20Less%20Secure-McCabe.pdf, accessed: 2018-04-01.

[35] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM Workshop on Quality of Protection, QoP 2008, Alexandria, VA, USA, October 27, 2008*, 2008, pp. 47–50. [Online]. Available: http://doi.acm.org/10.1145/1456362.1456372

[36] A. Abran, *Software Metrics and Software Metrology*. Wiley-IEEE Computer Society Pr, 2010.

[37] E. I. Oviedo, "Software engineering metrics i," M. Shepperd, Ed. New York, NY, USA: McGraw-Hill, Inc., 1993, ch. Control Flow, Data Flow and Program Complexity, pp. 52–65. [Online]. Available: http://dl.acm.org/citation.cfm?id=168026.168040

[38] "Cerebro," http://sites.google.com/site/cerebrofuzzer/, accessed: 2018-04-01.

[39] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *ICST*. IEEE Computer Society, 2012, pp. 121–130.

[40] P. D. Marinescu and C. Cadar, "KATCH: high-coverage testing of software patches," in *ESEC/SIGSOFT FSE*. ACM, 2013, pp. 235–245.

[41] A. D. Householder and J. M. Foote, "Probability-based parameter selection for black-box fuzz testing," 2012.

[42] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[43] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 345–362. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana

[44] W. Wang, H. Sun, and Q. Zeng, "Seededfuzz: Selecting and generating seeds for directed fuzzing," in *10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016*, 2016, pp. 49–56. [Online]. Available: https://doi.org/10.1109/TASE.2016.15

[45] "Discussion about aflgo in community," https://groups.google.com/forum/#!topic/afl-users/qcqFMJa2yn4, accessed: 2018-04-01.

[46] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[47] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.

[48] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 725–741. [Online]. Available: https://doi.org/10.1109/SP.2015.50

[49] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2313–2328. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134046

[50] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *CoRR*, vol. abs/1709.07101, 2017. [Online]. Available: http://arxiv.org/abs/1709.07101

[51] "American fuzzy lop - file format analyzer," https://github.com/mirrorer/afl/blob/master/afl-analyze.c, accessed: 2018-07-01.

[52] "libfuzzer," https://llvm.org/docs/LibFuzzer.html, accessed: 2018-04-01.

[53] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 579–594. [Online]. Available: https://doi.org/10.1109/SP.2017.23