Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference

Valentin J.M. Manès
CSRC, KAIST
Daejeon, Korea
valentinmanes@outlook.fr

Soomin Kim
KAIST
Daejeon, Korea
soomink@kaist.ac.kr

Sang Kil Cha
KAIST
Daejeon, Korea
sangkilc@kaist.ac.kr

ABSTRACT

Grey-box fuzzing is an evolutionary process, which maintains and evolves a population of test cases with the help of a fitness function. Fitness functions used by current grey-box fuzzers are not informative in that they cannot distinguish different program executions as long as those executions achieve the same coverage. The problem is that current fitness functions only consider a union of data, but not their combination. As such, fuzzers often get stuck in a local optimum during their search. In this paper, we introduce Ankou, the first grey-box fuzzer that recognizes different *combinations* of execution information, and present several scalability challenges encountered while designing and implementing Ankou. Our experimental results show that Ankou is 1.94× and 8.0× more effective in finding bugs than AFL and Angora, respectively.

CCS CONCEPTS

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

KEYWORDS

fuzz testing, guided fuzzing, grey-box fuzzing, software testing, principal component analysis

ACM Reference Format:
Valentin J.M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380421

1 INTRODUCTION

Fuzzing has recently gained popularity thanks to its proven record and its ease of use [37]. It has identified thousands of real-world vulnerabilities from a variety of software [6], and it has been developed by numerous security practitioners as well as academic researchers. Furthermore, it does not necessitate much information from the analyst beyond the entry point setup and optionally an initial set of test cases, so-called *seeds*.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380421

Seeds provide initial starting points for fuzzing. A seed corresponds to a program execution, and fuzzers can explore program paths mostly around this execution. Therefore, seeds need to be dynamically added or removed from the initial seed pool during a fuzzing campaign in order to efficiently explore the program state space. Grey-box fuzzers perform such a process by using a *fitness function*, which decides the quality of a given test case. As the seed pool evolves, fuzzers tend to generate more test cases that meet the fitness criteria enforced by the fitness function.

The current consensus is to leverage code coverage, such as branch coverage, as their fitness function. For instance, if a test case covers a new branch in the program under test, then we add it to the pool as it meets the fitness criterion. The actual implementation varies for each fuzzer, but they share the same idea: they prefer test cases that achieve new code coverage.

Despite its wide use, the current strategy of using code coverage as a fitness function suffers from critical information loss. Since code coverage only considers a *union* of information, if any one of the tested executions exercises a branch, for instance, then the branch is regarded as visited. As such, fuzzers can easily disregard test cases that do not improve code coverage even if they allow our fuzzers to exercise valuable execution paths. However, bugs often manifest when we exercise a specific execution path, but not when we visit a specific code snippet. For example, buffer overflow bugs do not occur when we visit the buggy loop, but they show up only when we exercise the loop more than a certain threshold.

Unfortunately, handing the aforementioned issue is challenging for the following three reasons: (**C1**) our fitness function should be *informative* in that it can quantify difference between program executions, (**C2**) our fitness function should be computationally *fast* while still being informative, and (**C3**) our fitness function should not accept too many seeds in the seed pool to be able to handle them in a practical manner.

First, our fitness function should be able to sensitively quantify program executions. That is, given two program executions, we need to be able to decide which one fits better for future fuzzing. Suppose we want to use path coverage as a fitness function. That is, if a test case exercises an unseen path, we consider it to meet the fitness criterion. In this case, the fitness function itself cannot judge the relative importance between test cases because the fitness function can only make a binary decision. The same problem exists for any coverage-based fitness function.

Second, computing informative fitness itself can be too costly. Since program executions naturally incorporate millions of instructions along with complex semantics, extracting their comprehensive information from an execution is typically an expensive process. Furthermore, the time complexity of a fitness function is critical

for grey-box fuzzing as we will have to invoke the fitness function for every test case generated during a fuzzing campaign.

Third, merely employing an informative fitness function can quickly make grey-box fuzzing unproductive as our fuzzer would admit too many seeds in the seed pool. For instance, one may produce a seed for every single path if we use path coverage as a fitness function. In this case, it may not even be possible for the fuzzer to give a trial for every seed in the pool.

In this paper, we tackle all the above challenges by introducing a novel fuzzing technique that we refer to as *distance-based fuzzing*. It leverages an informative fitness function that we call distance-based fitness function to deal with (**C1**). It also employs a novel dimensionality reduction technique that we call dynamic PCA to handle (**C2**). Lastly, it manages its seed pool with a technique called adaptive seed pool update for (**C3**).

Distance-based fuzzing employs an informative fitness function that we refer to *distance-based fitness function* to handle (**C1**). It measures the behavioral similarity between two executions by examining the *combinations* of exercised branches. The key intuition is to expand our view from a set of program elements (such as branches) to a set of *combinations* of program elements. By changing our perspective, we can easily identify the uniqueness of an execution in contrast to other executions even if the execution does not achieve novel code coverage. Note our fitness function only leverages readily available information in most state-of-the-art fuzzers, namely branch coverage (see §2.1).

Although the idea of distance-based fitness function integrates well with grey-box fuzzing, it is still challenging to adopt it in practice as computing the fitness itself is computationally expensive. This is mainly because we need to deal with a higher number of states as our fitness function gets more informative. According to our study, fuzzing with our distance-based fitness function makes fuzzers 13.2× slower.

To tackle this challenge (**C2**), we present dynamic PCA, which is inspired by a well-known statistical approach called Principal Component Analysis (PCA) [27]. PCA reduces the dimensionality of a data set while guaranteeing to preserve the maximum amount of information from the original set. However, PCA itself is computationally too expensive to be used with fuzzing. We cannot run PCA for every fuzzing iteration for the same reason the distance-based fitness cannot be directly used for fuzzing. To the best of our knowledge, none of the existing PCA variations suits our needs.

Therefore, we present a novel and practical dimensionality reduction technique that we call *dynamic PCA*. The core idea is to make the PCA computation to be incremental so that we do not need to recompute PCA from scratch. Our empirical study demonstrates that dynamic PCA can efficiently reduce the computational cost of the distance-based fitness function while introducing only 18% of information loss on average.

Finally, we introduce *adaptive seed pool update* to effectively manage the size of the seed pool (**C3**). The crux of our approach is to dynamically adjust the sensitivity of our pool update function based on the relative difference between program executions. Since our distance-based fitness function can quantify differences between program executions by its design, we can compare test cases based on their fitness to actively decide the sensitivity of the pool update function. In our study, Ankou without adaptive seed pool update was not functioning due to the excessive memory requirement and fitness computation cost.

To demonstrate our ideas, we designed and implemented Ankou, our prototype fuzzer, which leverages distance-based fitness function, dynamic PCA, as well as adaptive seed pool update to tackle all the three challenges. We performed a thorough evaluation for Ankou on 24 real-world application packages by spending a total of 2,682 CPU days. The results are promising, Ankou is 1.94× and 8.0× better in finding unique crashes compared to AFL [58] and Angora [14], respectively. Moreover, Ankou found a large variety of previously unknown software bugs in real-world software.

In summary, our contributions are as follows.

(1) We present an informative fitness function for grey-box fuzzing that we call distance-based fitness function.
(2) We introduce dynamic PCA, which is a novel approach to dynamically reduce the dimensionality of the distance-based fitness computation.
(3) We design and implement Ankou, the first fuzzer prototype for distance-based fuzzing.
(4) We create our own benchmark, which consists of 24 real-world application packages, and we make it public.
(5) We make our source code along with our benchmark public on GitHub to support open science: https://github.com/SoftSec-KAIST/Ankou.

2 BACKGROUND

This section presents fundamental concepts required to understand the proposed idea, and defines several necessary terminologies that we use throughout the paper.

2.1 Fitness and Local Optimum Problem

Current grey-box fuzzers primarily use code coverage as their fitness function: we add a test case to the seed pool if it achieves new code coverage. However, coverage-guided fuzzing strategies can miss out critical test cases that may guide fuzzers towards unseen execution paths while not necessarily improving the code coverage per se. We say we have reached a *local optimum* [30] as we cannot obtain any more test cases that fulfill our fitness criterion even though we have not yet tested all possible executions of the PUT.

This is certainly the case for fuzzing because some bugs can only be triggered when a specific execution path is exercised. For example, traditional buffer overflow bugs trigger when we exercise a loop more than a certain threshold, but not when we simply visited the loop; both the node and the branch coverage would remain the same.

To mitigate this problem, AFL [58] and its descendants [10, 11, 14, 32, 33] employ a modified version of branch coverage, which takes account of a hit count for each unique branch in the PUT. Note that the modified coverage can represent a greater number of program states compared to branch coverage: two executions may hit branches for different number of times while achieving the same branch coverage. We call such information gathered from every program execution by AFL as branch-hit-count state[1].

[1]Note that AFL introduces another approximation in their actual implementation: it bucketizes hit counts by powers of two to roughly measure how often each branch is

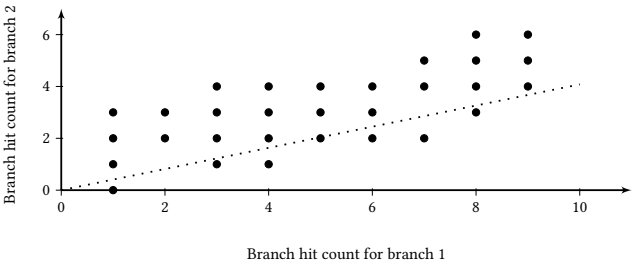


Figure 1: A hypothetical example showing branch-hit-count states of 30 unique program executions. Each dot represents a branch-hit-count state $\vec{x}$.

Definition 2.1 (Branch-Hit-Count State). Given a program p and an input t, the branch-hit-count state $\epsilon_p(t)$ is a vector

$$\epsilon_p(t) = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \vec{x}$$

where n is the number of branches in p, and x_i is the number of hits for branch i in the execution. For simplicity, we let the branch-hit-count state with a vector notation $\vec{x}$.

We note, however, that the fitness functions using branch-hit-count states still suffer from the local optimum problem. For example, let us consider a simple program p that has only three branches, and assume that three test cases t_1, t_2, and t_3 respectively produce the branch-hit-count states $\epsilon_p(t_1) = (1, 1, 2)$, $\epsilon_p(t_2) = (1, 1, 0)$, and $\epsilon_p(t_3) = (0, 1, 2)$. Suppose t_1 is firstly given, and t_2 and t_3 are produced while fuzzing the program. In this case, current fuzzers including AFL will favor t_1 as it can solely cover all the branches, and thus, t_2 and t_3 will be considered redundant, and will not be included in the population. Indeed, this is the key observation that motivates our research.

2.2 Principal Component Analysis

Principal Component Analysis (PCA) [27] is a way of reducing the dimensionality of a dataset while preserving as much information as possible. To understand the basic process of PCA, let us consider a hypothetical example where there is a program p with only two branches. Each execution of the program will produce a branch-hit-count state $\vec{x} = (x_1, x_2)$, which contains two hit-count numbers for each branch. Suppose our fuzzer has produced 30 test cases, which exercise 30 unique program executions. Figure 1 illustrates this example. Each dot represents a branch-hit-count state obtained by an execution, and the X- and Y-axis represent the hit count for branch 1 and 2, respectively.

The goal of PCA, in this example, is to obtain an 1-D plot from the 2-D plot in such a way that all the points in the resulting plot have the largest variance. For example, the dashed line in Figure 1 shows such an axis. If we project all the points onto the new axis, i.e., the dashed line, then we obtain the maximum possible variance between dots in the resulting 1-D plot.

exercised. We intentionally omit such details for brevity, but we note that it does not impact our analyses.

In this paper, we let PCA be a function that takes in a space representation as input, and returns an updated space representation as output. A space representation is a tuple of a basis $\mathbf{B}$ and a covariance matrix Σ. That is, PCA is a function of type

$$\text{PCA} : (\mathbf{B}, \Sigma) \rightarrow (\mathbf{B}', \Sigma').$$

The returned space representation has a reduced dimensionality and each axis, i.e., each column vector of $\mathbf{B}'$, is linearly independent to the other ones.

In the context of PCA, the tuple of a basis matrix and a covariance matrix effectively describes all the necessary information. A covariance matrix is a symmetric matrix representing how each data component are affected by each other. Since the example plot is on a 2-D Euclidean space, we can represent its basis as a 2 standard basis matrix.

$$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

To represent the relationship between the two components of Figure 1, we consider a 2×2 covariance matrix Σ of the space. Each element in the (i, j) position is the covariance between the i-th and j-th components.

$$\Sigma = \begin{bmatrix} 6.930 & 2.728 \\ 2.728 & 2.231 \end{bmatrix}.$$

The element in (1,1) represents the covariance between the first component and itself, which means the variance of the first component. The elements in (1, 2) and (2, 1) are the same as the covariance between two components is the same regardless of their order.

In the perspective of linear algebra [52], PCA is equivalent to an eigendecomposition process on the given covariance matrix, which returns a diagonal matrix Σ' and a basis matrix $\mathbf{B}'$. The basis $\mathbf{B}'$ contains eigenvectors as its column vectors, which represents the axes of a new coordinate system. Σ' has the eigenvalues on its diagonal entries, which are the variances of the corresponding new axes. Specifically when applied to Σ above, we obtain the following covariance matrix and basis matrix.

$$\Sigma' = \begin{bmatrix} 8.180 & 0 \\ 0 & 0.981 \end{bmatrix}, \text{ and } \mathbf{B}' = \begin{bmatrix} 0.909 & 0.417 \\ 0.417 & -0.909 \end{bmatrix}.$$

In order to maximize the variance of the lower dimensionality space, we chose the axes with the highest variances. In this case, since it has a variance of 8.180, we select the vector $[0.909, 0.417]$, which corresponds to the dashed line of Figure 1. This becomes our new axis of the desired 1-D plot.

3 DISTANCE-BASED FUZZING FITNESS

The key challenge that we address here (**C1**) is designing an informative fitness function for grey-box fuzzing, which can sensitively quantify the difference between test cases and their corresponding executions on the PUT. Remarkably, we found that the branch-hit-count states used by current fuzzers already provide just enough information about test cases for judging their *potential* to be used as a future seed. The idea is to consider each branch-hit-count state as a *vector*, as defined in §2.1, which enables us to compute relative distances between them.

Since we are dealing with relative distances, two distinct executions that achieve the same coverage, but produce different branch-hit-count states would represent two unique vectors in the

space, and we can naturally quantify their difference compared to the other vectors in the space. Indeed this is the key intuition of our distance-based fitness function.

3.1 Fitness as Distance between Vectors

In our model, a branch-hit-count state corresponds to a vector in a space that we call the branch-hit-count state space Ω_p, which is formally defined as follows. For any test case t, we can obtain a branch-hit-count state $\epsilon_p(t)$ in Ω_p by executing p with t.

Definition 3.1 (Branch-Hit-Count State Space). Given a program p, the *branch-hit-count state space* of p, Ω_p, is the set of all possible branch-hit-count states we can obtain by executing p.

With this, we now introduce the concept of *execution distance*, which measures the relative distance between two branch-hit-count states, thereby determining the difference between their two executions. Note that the distance between two vectors is dependent on which space we are in. Thus, our definition of execution distance takes a basis **B** into account.

Definition 3.2 (Execution Distance). Given a program p and a basis **B**, any pair of branch-hit-count states in Ω_p have an execution distance $\delta_{\mathbf{B}}$ on the space defined by **B**, which is simply defined as the Euclidean distance

$$\forall(\vec{x}, \vec{y}) \in \Omega_p^2,\ \delta_{\mathbf{B}}(\vec{x}, \vec{y}) = \|\vec{x}^T\mathbf{B} - \vec{y}^T\mathbf{B}\|.$$

Intuitively, two executions are similar to each other when their execution distance is small, and vice versa. For example, suppose there is a program with only three branches, and there are three executions of the program, which result in the branch-hit-count states $\vec{x} = (3, 0, 1)$, $\vec{y} = (3, 0, 0)$, and $\vec{z} = (0, 1, 1)$, respectively. In this case, we can readily determine that $\vec{x}$ and $\vec{y}$ are similar to each other as the first branch is exercised three times in both cases, unlike $\vec{z}$. Although branch hit counts do not completely reflect the semantics of the program executions, we can still extract meaningful distinction between executions.

Since a test case produces an execution for a given PUT, we can compare two test cases for the PUT by leveraging their execution distances. That is, the execution distance allows us to compute the difference between given test cases with respect to the PUT. Therefore, we devise a new fitness function to quantify the *novelty* of a given test case compared to the current population, i.e., test cases in the seed pool. Let $\Pi = \{t_1, t_2, \cdots, t_m\}$ be a seed pool of m test cases, the *distance-based fitness* of a newly generated t is then the minimum execution distance between t and all the seeds in Π. As execution distance can vary depending on the current space we are in, the definition of distance-based fitness function also takes the current space (**B**) into account.

Definition 3.3 (Distance-based Fitness Function). Given a program p and a basis **B**, the distance-based fitness $\Delta_{\mathbf{B}}(t, \Pi)$ of a test case t with regard to a seed pool Π is the minimum execution distance between $\epsilon_p(t)$ and a set $\{\forall i \in \Pi : \epsilon_p(i)\}$ on the space defined by **B**. Formally, the distance-based fitness function is

$$\Delta_{\mathbf{B}}(t, \Pi) = \min_{i \in \Pi} \delta_{\mathbf{B}}(\epsilon_p(t), \epsilon_p(i)).$$

With the distance-based fitness function, we can now quantify the difference between a test case and a pool of test cases. For example, let us consider the test cases with the following states: $\epsilon_p(t_1) = (1, 1, 2)$, $\epsilon_p(t_2) = (0, 1, 1)$, $\epsilon_p(t_3) = (1, 0, 2)$, and $\epsilon_p(t_4) = (0, 3, 3)$. Assume the current seed pool contains the first two seeds $\{t_1, t_2\}$, and our fuzzer has generated the test case t_3 and t_4. We can now compare the two test cases, decide which is the *fittest* and include it in the pool. Using the standard basis as **B**, we obtain $\Delta_{\mathbf{B}}(t_3, \{t_1, t_2\}) = 1$, and $\Delta_{\mathbf{B}}(t_4, \{t_1, t_2\}) \approx 2.45$. Since t_4 execution is further away from the seed pool, its inclusion in the pool brings more novelty than t_3 would. Hence, t_4 is a better fit. Although neither test cases bring new coverage, this is representative of how the combination of their branch-hit-count differs. t_3 execution only differs from t_1 on the second branch, while t_4 differs from t_1 on all the branches.

3.2 Impracticality of Distance-based Fitness

Our distance-based fitness function is indeed informative—it provides a way to effectively quantify the fitness of generated test cases—but such benefit comes with a price. Although it does not require any new coverage metric to be introduced, it is not feasible to apply the idea directly to current fuzzers due to its high computational cost.

Since we have to compute the fitness for every generated test case, the performance of fitness computation is critical. Unfortunately, its time complexity is roughly $O(mn)$, where m is the number of seeds in our seed pool, and n is the number of branches in the PUT. Note that there can be easily thousands of seeds in the population as well as thousands of branches in the PUT. This is indeed the challenge (**C2**) that we address in the paper. To make our approach practical, we need to reduce one of the terms.

One plausible way to improve the performance is to employ a specialized data structure designed for efficient distance queries, such as M-tree [16]. It allows us to reduce the number of seeds to look for without any loss of information. However, it does not guarantee any practical lower bound. In a preliminary study we performed, we only observed about 70% of performance improvement with M-tree, which was far from enough to make the distance-based fitness function practical for fuzzing.

Therefore, we address this impracticality challenge by reducing the dimensionality of the branch-hit-count states, which effectively reduces n, the number of branches to consider. Recall that PCA (as introduced in §2.2) is a well-known methodology for reducing the dimensionality of a dataset while minimizing the loss of information. The goal here is to reduce the dimensionality of the branch-hit-count state space, which can drastically improve the performance of our fitness function computation. Nevertheless, this new intermediary step introduces its own challenges as we describe in the following section.

4 DYNAMIC PCA

The high computational demand of distance-based fitness function naturally leads us to the second challenge (**C2**). To tackle the challenge, we introduce *dynamic PCA*, a novel technique for reducing the dimensionality of the branch-hit-count states.

As the name implies, dynamic PCA is inspired by PCA, but PCA itself does not perfectly meet our needs for several reasons. First,

PCA itself is computationally expensive: it has a cubic-time complexity in the number of samples and dimensions [21]. Second, the underlying probability distribution we are sampling from changes every time as the seed pool varies: grey-box fuzzing creates a dynamic environment. This means we would need to compute PCA every time we add a new seed to our pool, but its cost would become extremely high. As we will explain in §8, several variations of PCA have been introduced, none of them suits our case. Dynamic PCA overcomes these challenges by presenting an efficient approximation of PCA, which eventually enables the practical use of the distance-based fitness for grey-box fuzzing.

4.1 Algorithm Overview

At a high level, *dynamic PCA* achieves its performance improvement by (1) reducing the number of times eigendecomposition is run, and (2) limiting the number of axes in the space to perform eigendecomposition on. Dynamic PCA periodically runs eigendecomposition, i.e., the standard PCA, for every time interval we choose, which is one minute in the current implementation. It maintains a reduced covariance matrix and incrementally updates the matrix. When we perform eigendecomposition, we run it only on the reduced matrix.

The DynPCA function in Algorithm 1 describes the main algorithm, which roughly takes in a space representation ($\mathbf{B}$, Σ), and returns an updated one. The initial space representation is obtained by running the standard PCA for the seeds in the initial seed pool given by the user. Unlike the standard PCA we described in §2.2, though, it also takes in three more parameters as input: (1) $\vec{x}$ is the branch-hit-count state obtained by executing the currently generated test case, which is required to update the space information; (2) s is the number of generated test cases; and (3) θ_{exp} is a variable automatically set by our algorithm, whose initial value is the infinity ∞. Dynamic PCA operates with three major functions: ExpandBasisIfInteresting, UpdateCovMatrix, and PeriodicDecompose.

- **ExpandBasisIfInteresting** checks whether the branch-hit-count state $\vec{x}$ suffers a large information loss when projected on $\mathbf{B}$. If it does, then we consider $\vec{x}$ as an "interesting" vector, and add it to our basis $\mathbf{B}$ as an extra axis (see §4.2).
- **UpdateCovMatrix** updates the current covariance matrix Σ with regard to the given branch-hit-count state $\vec{x}$ (see §4.3). Note that Σ effectively summarizes the branch-hit-count states of all the test cases the fuzzer observed so far.
- **PeriodicDecompose** periodically readjusts the basis $\mathbf{B}$ every minute by running the standard PCA. The current time interval is empirically chosen, but it is a user configurable parameter in our implementation. Note that this function needs to handle only a reduced space returned by the previous steps. That is, the number of axes in $\mathbf{B}$ is several orders of magnitude smaller than the total number of branches in the PUT. This is indeed the key to our approach.

Information Loss due to Dynamic PCA. Although dynamic PCA makes it efficient to compute the principal components of a given space representation, it loses the guarantee of maximizing the variance of the reduced space. Nonetheless, our empirical result shows that the information loss caused by dynamic PCA is 20% or less in most subjects we tested (see §6.3). Therefore, dynamic PCA can be a practical alternative to standard PCA.

Algorithm 1: Dynamic PCA

```
   // θ_exp is globally given, and initially set to ∞.
1  function ExpandBasisIfInteresting(B, Σ, x⃗)
2      loss ← sqrt(‖x⃗‖² − ‖x⃗ᵀB‖²) // By Pythagoras
3      if loss > θ_exp then
4          B, Σ ← Append(B, Σ, x⃗)
5          B ← GramSchmidt(B)
6      θ_exp ← UpdateLoss(θ_exp, loss)
7      return B, Σ
8  function PeriodicDecompose(B, Σ)
9      if IsOneMinutePassed() then
10         B, Σ ← PCA(B, Σ)
11     return B, Σ
   // The main function
12 function DynPCA(B, Σ, x⃗, s)
13     B, Σ ← ExpandBasisIfInteresting(B, Σ, x⃗)
14     Σ ← UpdateCovMatrix(B, Σ, x⃗, s)
15     B′, Σ′ ← PeriodicDecompose(B, Σ)
16     return B′, Σ′
```

4.2 Incremental Basis Expansion

In ExpandBasisIfInteresting, the information loss caused by projecting the execution on $\mathbf{B}$ is quantified by `loss` in Line 2. In Line 3, the loss is considered significant if above the threshold θ_{exp}. Then, in Line 4, the new branch-hit-count state, which is a vector by definition, is appended to the basis and the covariance matrix Σ is expanded by one row and one column. The new basis $\mathbf{B}$ is then orthonormalized by Gram-Schmidt [53] in Line 5.

To get a better understanding, let us consider the previous example illustrated in Figure 1. There are only two branches in the program and we have 30 initial seeds, one for each point in the plot. The initial $\mathbf{B}$ and Σ are set using the standard PCA on the initial seeds. Note that this is an expensive operation as the branch-hit-count state is likely to have many dimensions. We can only afford it once at the initialization. Now, let us assume that the first test case we generate manifests the branch-hit-count state $\vec{x} = (1, 100)$. We pass the state to DynPCA, and we reach the ExpandBasisIfInteresting function. The branch-hit-count state is indeed a large outlier, which will pass the test in Line 3 of ExpandBasisIfInteresting. Therefore, this vector will be appended to $\mathbf{B}$, and then orthonormalized into $\left[-0.417, 0.909\right]$ by the Gram-Schmidt [53] procedure. The following test cases and their branch-hit-count states will be projected onto the new 2-D basis until the call to the PCA function in PeriodicDecompose will reconsolidate $\mathbf{B}$ into a single vector.

Additionally, ExpandBasisIfInteresting updates the expansion threshold θ_{exp} in Line 6. UpdateLoss records all the `loss` values in the past minute and sets θ_{exp} to the maximum of the held data. In our experiments, this was enough to make dynamic PCA maintain a sufficient number of new axes while keeping the computational cost low. Optimizing UpdateLoss is beyond the scope of this paper.

4.3 Dynamically Updating Covariance Matrix

As we generate test cases, we should also incrementally update our covariance matrix to take account of newly sampled test cases

added to our space. However, there is an issue to be addressed for updating our covariance matrix. PCA assumes that our sampling process is performed on a constant probability distribution, but this is not the case for grey-box fuzzers where a change in the seed pool implies a change in the sampling process.

To address this problem, we implement UpdateCovMatrix to include a *discount factor* α in order to favor newer test cases rather than older ones. Particularly, every time we update Σ, we give higher weights to newer test cases by progressively decreasing weights to the previous covariance matrix. Formally, given the $(s+1)$-th generated test case, which produces $\vec{x}$, the UpdateCovMatrix operates by updating Σ' as follows.

$$\Sigma' = \frac{(\vec{x}^T\mathbf{B}) \cdot (\vec{x}^T\mathbf{B})^T + \alpha w_s \Sigma}{1 + \alpha w_s}, \text{ where } w_s = 1 + \alpha + \alpha^2 + \cdots + \alpha^{s-1}.$$

If α is set to 0, then we completely ignore the history, and we end up solely using the latest test case to construct Σ'. On the other hand, when $\alpha = 1$, the term αw_s becomes s, and the resulting formula simply represents an incremental mean where s is the total number of elements. When α is between zero and one, we effectively give a weight of αw_s to the old covariance matrix Σ in order to give the decreasing influence to it as time passes. The lower α is, the more we forget about the past. Note that the old covariance matrix represents s total test cases generated so far where the first test case has a weight $\alpha^{(s-1)}$. We empirically set α to $1 - 10^{-6}$ in our current implementation. Although not explicitly mentioned for brevity, the branch-hit-count state $\vec{x}$ is centered before being projected on $\mathbf{B}$.

5 DISTANCE-BASED FUZZING

In this section, we first show a way to dynamically adjust the sensitivity of our fitness function to handle (**C3**). The primary issue here is that an informative fitness function such as our distance-based fitness function would accept too many seeds in the pool. To set the sensitivity of the fitness function, we introduce adaptive seed pool update, a novel population update mechanism that dynamically changes its fitness criterion. With this, we present the design and implementation of Ankou, our fuzzer prototype that enables distance-based fuzzing by addressing all three challenges (**C1**, **C2**, and **C3**). Ankou leverages the distance-based fitness function (see §3) to obtain *informative* feedback, and employs the dynamic PCA (see §4) to efficiently compute the distance-based fitness function. It also uses adaptive seed pool update to dynamically changes its fitness criterion.

5.1 Adaptive Seed Pool Update

The distance-based fitness of a test case characterizes its novelty compared to the current population, but having a way to measure novelty (or fitness) does not tell us when should we add our test case to the seed pool. Of course, we can add our test case to the pool when its distance-based fitness is above a threshold, but what should be the value of the threshold then? Note that the choice of this threshold is critical as it sets the sensitivity of a fuzzer to new behaviors of the PUT. If it is infinitely high, the seed pool is constant and the population does not evolve. On the other hand, if the threshold is set to zero, any test case will be added to the pool, which can quickly pack the seed pool.

Algorithm 2: Adaptive seed pool update.

```
  // The space information (B, Σ) is globally given.
  // θ_fit is globally given, and initially set to zero.
1 function PoolUpdate(t, s, Π)
2     B, Σ ← DynPCA (B, Σ, ε_p(t), s)
3     if Δ_B(t, Π) > θ_fit then
4         Π' ← AddToPool(t, Π)
5         θ_fit ← min_{i∈Π'} Δ_B(i, Π' \ {i})
6         return Π'
7     else return Π
```

Thus, we propose *adaptive seed pool update*, a novel technique that dynamically selects the threshold to adaptively control the sensitivity of our fuzzer. The PoolUpdate function in Algorithm 2 describes the overall algorithm, which takes in a newly generated test case t, the total number of test cases generated so far s, and the seed pool Π as input. It then outputs an updated seed pool Π'. In Line 2, we perform dynamic PCA in order to make our fitness function computation $\Delta_{\mathbf{B}}$ efficient. We then check if the distance-based fitness is bigger than the fitness threshold θ_{fit}, which is initially given as zero. If so, we update both the seed pool and the threshold in Line 4 and 5. The AddToPool function in Line 4 first pops out the seed with the lowest distance to the population, and then add our test case t to the pool. That is, we remove the least fit test case from the pool while adding a new one. To maintain a sufficient amount of seeds, AddToPool will only remove a test case when the pool has 1,000 or more seeds in our current implementation.

In Line 5, we compute the distance-based fitness for all the seeds in the pool, and set the current minimum fitness as a new threshold. The intuition here is that in order for a test case to be useful, it should be at least further away from the pool than the smallest gap between the seeds. More formally, we set the next threshold θ_{fit} by

$$\theta_{\text{fit}} = \min_{i \in \Pi} \Delta_{\mathbf{B}}(i, \Pi \setminus \{i\}) = \min_{(i_1, i_2) \in \Pi^2, i_1 \neq i_2} \delta_{\mathbf{B}}(\epsilon_p(i_1), \epsilon_p(i_2)).$$

Practical Impact of Adaptive Seed Pool Update. To understand the impact of adaptive seed pool update, we performed a preliminary study where we ran Ankou without adaptive seed pool update on a subject in our benchmark. As a result, Ankou was killed by the OS due to its excessive memory use after a few minutes. During its run, Ankou was mostly spending its time computing $\Delta_{\mathbf{B}}$ in Line 3. Since there are too many seeds in the pool, its computational cost, even with dimensionality reduction, became too extreme to be able to run in practice. Thus, we conclude that adaptive seed pool update is an essential piece of distance-based fuzzing.

5.2 Ankou Architecture

Ankou follows the general architecture of grey-box fuzzing, which consists of three major components: seed scheduler, pool manager, and tester. Figure 2 illustrates the overall design of Ankou. The seed scheduler selects a seed for fuzzing and passes it to the tester module. The tester then generates inputs by mutating the given seed and run the PUT. Upon the PUT execution, the tester passes the execution trace to the fitness function of the pool manager, which computes its fitness value. The PoolUpdate function in the

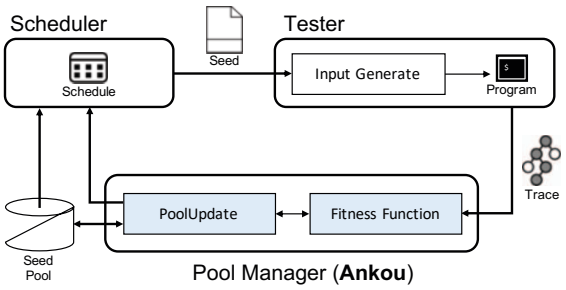


Figure 2: Ankou architecture.

pool manager module then updates this value with the adaptive seed pool update technique.

Note that the only difference between existing grey-box fuzzers and Ankou is in the design of the pool manager module. Particularly, Ankou uses the adaptive seed pool update for pool update, and the distance-based fitness function enabled by the dynamic PCA. Any grey-box fuzzers can easily benefit from distance-based fuzzing.

5.3 Implementation

Ankou is built upon our own AFL implementation in Go [3]. It is a simplified version of AFL, which implements most of the AFL's features, but not all. For instance, our implementation does not include culling since Ankou performs it on-the-fly in Line 4 of Algorithm 2. Ankou does not implement the seed prioritization heuristics employed by AFL, where seeds having fast throughput and high coverage are likely to get a higher priority. Since our goal in this paper is on designing a new fitness function for grey-box fuzzing, we intentionally omitted such heuristics in our implementation to effectively measure the impact of our fitness function. Instead, Ankou chooses seeds from the seed pool at random and generates test cases for a constant time interval, which is currently one second in the current implementation.

Our current implementation of Ankou consists of 8K lines of Go (as measured by CLOC [18]). We used the Gonum numeric library [4] in order to implement the PCA function. Ankou employs the same instrumentation module provided by the vanilla AFL [58]. Therefore, Ankou can easily support ASan [49] and AFL-lafintel [29]. We make our prototype implementation as well as our benchmark publicly available on GitHub [38].

6 EVALUATION

We evaluated Ankou on the following research questions.

(1) How much was the speed gain enabled by dynamic PCA and what was its impact on bug discovery?
(2) Can dynamic PCA effectively reduce space dimensionality without significant information loss?
(3) How does the distance-based fitness function compare to coverage-based fitness function?
(4) How much is the computational cost of distance-based fuzzing?
(5) How does Ankou compare to other grey-box fuzzers?

6.1 Experimental Setup

Basic Setup. We performed our experiments on two server machines, each of which is equipped with 44 Intel Xeon E5-2699 v4 cores and 512GB of RAM. For every fuzzing campaign, we used a Docker container assigned to a single core. Unless stated otherwise, all the reported numbers are the average of six repeated fuzzing campaigns, each of which was performed for 24 hours. We used the Mann-Whitney U-Test [7] with $\alpha = 0.05$ to determine the significance of each experiment. When we report the number of unique crashes, we follow AFL's definition: if two crashes achieve the same branch coverage, we count them as one.

Measuring Throughput. In RQ1, RQ3, and RQ4, the test case generation throughput—the number of test cases the fuzzer produced per second—is used as a proxy to measure the cost of the analysis each fuzzer performs. When a fuzzer performs a time consuming operation (in the case of Ankou, the dynamic PCA), it is at the expense of the test case that could have been generated and run in the same amount of time. Thus, the lower the throughput is, the higher the cost of the analysis the fuzzer is performing.

Fuzzers to Compare. Since Ankou is a source-based fuzzer it cannot be fairly compared to binary-based fuzzers such as Eclipser [15] or RedQueen [9]. Recent source-based fuzzers such as Steelix [33] and CollAFL [22] were not made available for comparison. LibFuzzer [5] requires a custom library caller to be made to run experiments. Hence, we compare Ankou against AFL 2.52b [58], the latest version at the time of writing, and Angora [14]. When we run AFL, we used the "`-d`" option, which essentially enables AFLFast [2, 11].

Our Benchmark. To create our benchmark, we collected all the programs, but with the latest versions, used by CollAFL [22]. This benchmark includes a total of 24 different program packages, constituting more than 5 MLoC (see Table 1). When a program package contains more than one executable, we consider all of them as a separate subject. For example, `libtasn1` is a library, which has three distinct wrapper program executables in its source distribution. In this case, we regard each executable as a distinct *subject*[2]. As a result, we obtained 150 different subjects from the 24 packages. Since the authors of CollAFL have not opened their benchmark to the public, we obtained initial seeds by gathering test cases provided by each package, and we did not perform any additional processing. We make our benchmark public along with the source code.

Hours of Experiments. We ran Ankou and AFL on each subject of our benchmark suite for 24 hours, and repeated this experiment for 6 times. We did the same for Angora, but only on the subjects it was successfully compiled for (see §6.6). To answer RQ1 and RQ3, we selected 24 subjects from the benchmark by randomly choosing one executable per package. We then ran 24-hour fuzzing for each of the 24 subjects of the selected subset 6 times. In total, all our experiments constitute 2,682 CPU days.

6.2 RQ1: Impact of Dimensionality Reduction

Does dynamic PCA really help improve the efficiency of distance-based fitness function? To answer this question, we run Ankou in two modes: (mode 1) Ankou with the distance computed using the

[2]The term subject is widely used in practice by LibFuzzer [5].

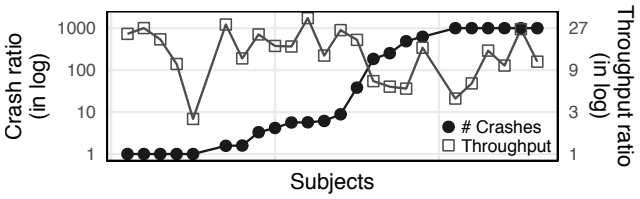


Figure 3: Comparison between distance-based fitness function with and without dynamic PCA in terms of the number of crashes found and test case generation throughput.

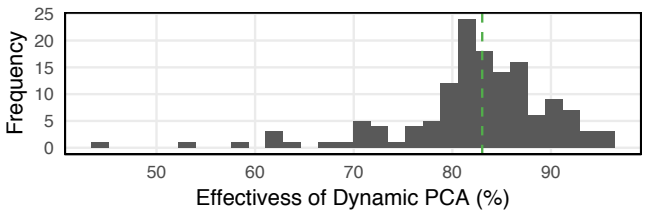


Figure 4: The effectiveness of dynamic PCA represented by the percentage of preserved information on 150 subjects. The green dashed line represents the median effectiveness.

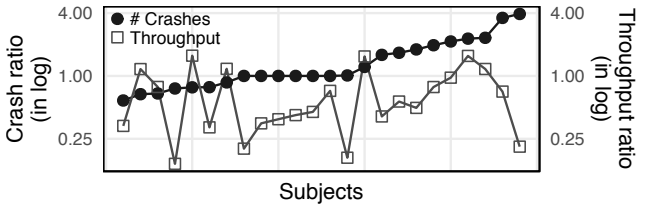


Figure 5: Comparison between distance-based fitness and coverage-based fitness function in terms of the number of crashes found and test case generation throughput.

dynamic PCA, and (mode 2) Ankou with distance-based fitness function but without dynamic PCA. Meaning, the distance is computed using Definition 3.2, without any dimensionality reduction.

Figure 3 illustrates our experimental results after 24 hours of fuzzing on each of the 24 selected subjects. The line with circles shows the ratio between the number of crashes found with mode 1 and 2. The line with squares shows the ratio between the test generation throughput of Ankou with mode 1 and 2. The first five crash points (circles) have the ratio of one as we found no crash in both cases. The last six points, with a ratio of 1,000, are the cases where Ankou in mode 1 found crashes while mode 2 did not. In all cases, the crash ratio was higher than one, meaning that using dynamic PCA always produces better results than using Defintion 3.2. Ankou found 11.8× more unique crashes and generated 13.2× more test cases with dynamic PCA than without it.

The "①<③ U value" and the "②<④ U value" columns of Table 1 describe the result of the Mann-Whitney U Test on the experiment. A value written in bold and with a grey background means the experiment was successful. If the value is close to 1.0, it means the hypothesis is validated, e.g. "①<③". On the other hand, if the value is close 0.0, it means the opposite is validated, e.g. "①>③". For those subjects that show statistical significance, dynamic PCA gave considerably better results in terms of both bug finding and throughput. These results confirm the necessity of our dynamic PCA to enable the practical usage of the distance-based fitness function.

6.3 RQ2: Effectiveness of Dynamic PCA

Although dynamic PCA allows us to efficiently generate test cases, it comes at a price. Since dynamic PCA is an approximation process, it may suffer from a loss of information. If so, how much would be the loss? In other words, do the identified basis from dynamic PCA successfully maximize the variances of branch-hit-count states?

To answer the question, we measured the effectiveness of dynamic PCA on all the fuzzing campaigns against the 150 subjects in our benchmark. The effectiveness is quantified by the variances, i.e., eigenvalues, appeared in the resulting covariance matrix Σ'. By computing the portion of the variances of the selected axes in Σ', we can quantify how much information is lost by running the dynamic PCA (or standard PCA) process. For instance, if we look back at the example in §2.2, the effectiveness of PCA was about 89% (= 8.18/(8.18 + 0.981)). The closer this number is to 100%, the less loss of information in the PCA computation will be.

Figure 4 is the histogram showing the effectiveness of dynamic PCA for all the 150 subjects. For 80% of the subjects, the effectiveness was above 78.8%, and for 90% of the subjects, the effectiveness was above 72.7%. In other words, dynamic PCA was able to keep 80% of the information obtained from program executions in most of the subjects. This result indeed highlights the key contribution of our paper: dynamic PCA can effectively reduce the dimensionality of program state space without losing much information, which enables the practicality of distance-based fuzzing.

6.4 RQ3: Distance- vs. Coverage-based Fitness

Recall from §3, one of the key motivations of distance-based fitness function was that coverage-based fitness functions do not provide sufficient information to fuzzers for finding bugs. To confirm the value of distance-based fitness function, we ran Ankou with and without distance-based fitness function. When disabling the distance-based fitness function for Ankou, we only used branch coverage as its fitness criterion.

Figure 5 shows our results after 24 hours of fuzzing on the same 24 subjects as in RQ1. Overall, distance-based fitness function found 1.5× more unique crashes, and produced more crashes in 60% of the subjects, excluding the 5 subjects where no crashes were found. At best, the distance-based fitness function found 4× more unique crashes than without it. As the right-most columns of Table 1 describe, for those subjects that manifest statistic significance, distance-based fitness function gave better results for 83% of the cases (5 out of 6). However, for the other subjects that were not found to be statistically significant, the distance-based fitness function was usually still an improvement over the traditional coverage-based fitness function in terms of the number of crashes found. Otherwise, the difference was negligible. Since the expected loss is slim while the potential gain is large, we should benefit from choosing the distance-based fitness function a priori.

Table 1: Detailed experimental results for RQ1 and RQ3. We show the Mann-Whitney U test results along with the detailed numbers for every experiment we performed. The shared areas indicate statistically significant results.

					No PCA vs. Dynamic PCA (RQ1, §6.2)				Coverage vs. Distance-based Fitness (RQ3, §6.4)			
Package Name	**Version**	**KLoC**	**① Ankou Crashes**	**② Ankou Throughput**	**③ No PCA Crashes**	**① < ③ U Value**	**④ No PCA Throughput**	**② < ④ U Value**	**⑤ Cov-based Crashes**	**① < ⑤ U Value**	**⑥ Cov-based Throughput**	**② < ⑥ U Value**
binutils	2.32	1687	0.167	102	0 (-Inf)	0.42	23.9 (÷4.27)	**0.00**	0.167	0.50	508 (+397%)	**1.00**
bison	3.3	82.4	497	20	13 (÷38.22)	**0.00**	1.01 (÷19.86)	**0.00**	232 (-53%)	**0.00**	20.8 (+3%)	0.33
catdoc	0.95	3.8	28.7	56.7	8.67 (÷3.31)	**0.00**	2.48 (÷22.90)	**0.00**	23.5 (-18%)	0.47	36.9 (-34%)	**0.11**
cflow	1.6	37.8	470	78.9	83.3 (÷5.64)	**0.00**	4.72 (÷16.70)	**0.00**	262 (-44%)	**0.00**	160 (+102%)	0.69
clamav	0.101.2	840	211	89.3	37 (÷5.70)	**0.00**	2.55 (÷35.07)	**0.00**	91 (-56%)	**0.00**	76.6 (-14%)	0.25
GraphicMagick	1.3.31	252	13.7	66.6	0 (-Inf)	0.08	4.44 (÷14.99)	**0.00**	3.8 (-72%)	0.35	94.1 (+41%)	**1.00**
jasper	2.0.14	30.8	324	294	36.7 (÷8.84)	**0.00**	11.5 (÷25.64)	**0.00**	142 (-56%)	**0.00**	189 (-35%)	0.25
libav	12.3	586	23.7	14.8	5.67 (÷4.18)	**0.00**	0.872 (÷16.94)	**0.00**	35.4 (+49%)	0.80	12.7 (-14%)	0.40
dwarf	b4f198	93.8	15.2	119	9.67 (÷1.57)	**0.00**	4 (÷29.69)	**0.00**	17.5 (+15%)	**0.92**	102 (-14%)	0.44
libexiv2	0.27.1	72.9	57.3	49.1	36 (÷1.59)	0.50	4.02 (÷12.22)	**0.00**	84.4 (+47%)	0.67	62.6 (+27%)	**0.90**
libgxps	0.3.1	8.8	2.33	48.6	2.33	0.50	19.4 (÷2.51)	**0.00**	3 (+28%)	0.83	31.1 (-36%)	**0.00**
liblouis	3.9.0	36.2	488	30.9	1 (÷487.67)	**0.00**	5.57 (÷5.54)	**0.00**	124 (-74%)	**0.00**	147 (+375%)	**1.00**
libming	0.4.8	81.2	337	56.2	1.33 (÷252.88)	0.38	9.63 (÷5.83)	**0.00**	445 (+31%)	0.58	390 (+594%)	**1.00**
mpg123	1.25.10	41.1	0	18	0	0.50	0.894 (÷20.14)	**0.00**	0	0.50	42.7 (+137%)	**1.00**
libncurses	6.1	112	209	33	34 (÷6.14)	0.38	2.5 (÷13.16)	**0.00**	359 (+71%)	0.56	99.2 (+200%)	**1.00**
libraw	0.19.2	51.3	17.2	58.8	0 (-Inf)	0.33	9.25 (÷6.36)	**0.00**	22 (+28%)	0.50	183 (+210%)	**1.00**
libsass	3.5.2	24.7	5	95.5	0 (-Inf)	0.33	3.63 (÷26.30)	**0.00**	3 (-40%)	0.75	168 (+75%)	**1.00**
libtasn1	4.13	30.3	0	78.4	0	0.50	3.37 (÷23.25)	**0.00**	0	0.50	204 (+159%)	**1.00**
libtiff	4.0.10	67.6	0.167	117	0 (-Inf)	0.42	10.4 (÷11.22)	**0.00**	0.167	0.50	259 (+121%)	**1.00**
libtorrent	1.2.1	119	0	96.7	0	0.50	3.58 (÷27.04)	**0.00**	0	0.50	134 (+38%)	**0.89**
nasm	2.14.03rc2	94.0	46.3	30.4	0 (-Inf)	**0.00**	3 (÷10.14)	**0.00**	45.8 (-1%)	0.61	185 (+507%)	**1.00**
pspp	1.2.0	257	312	19	0.5 (÷623.33)	0.29	1.18 (÷16.17)	**0.00**	196 (-37%)	0.50	46.5 (+144%)	**1.00**
tcpdump	4.9.2	77.3	0	66.3	0	0.50	6.27 (÷10.57)	**0.00**	0	0.50	189 (+184%)	**1.00**
vim	8.1.1332	347	123	10.2	0.667 (÷185.00)	**0.00**	1.51 (÷6.74)	**0.00**	62.7 (-49%)	0.33	13 (+27%)	0.53
Total		5037	3180	1649	269 (÷11.82)		139.7 (÷11.80)		2152 (-32%)		3352 (+103%)	

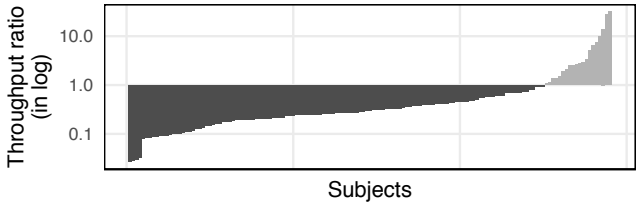


Figure 6: Comparison of test case generation throughput between Ankou and AFL.

On the other hand, Ankou using distance-based fitness function had a test case generation throughput 51% lower because of the time spent on computing its fitness function. Thus, even if the distance-based fitness incurs a significantly slower throughput, it allows Ankou to find more unique crashes.

Remarkably, the difference in branch coverage was *insignificant*: it was under 1.5% on average. This result coincides with our observation: software bugs do not manifest when we achieve certain code coverage, but when we exercise a specific execution path. Therefore, we conclude that distance-based fitness function benefits grey-box fuzzing in terms of finding software bugs in an effective manner.

6.5 RQ4: Distance-based Fuzzing Cost

In this subsection, we evaluate the practicality of distance-based fuzzing with the following two questions: (1) Is the dynamic PCA necessary? How slow would it be if we were to use the standard PCA instead?; and (2) How much performance overhead can we observe by enabling distance-based fuzzing assisted by dynamic PCA instead of a coverage-based approach?

The answer to the first question is indeed simple: our initial fuzzer prototype with the standard PCA was not usable as it spends most of its time on computing the PCA. On our machine, it took about an hour to compute the PCA for 5,000 seed files. Given that fuzzers typically run thousands of test cases per second, it would not be possible to use the standard PCA in practice.

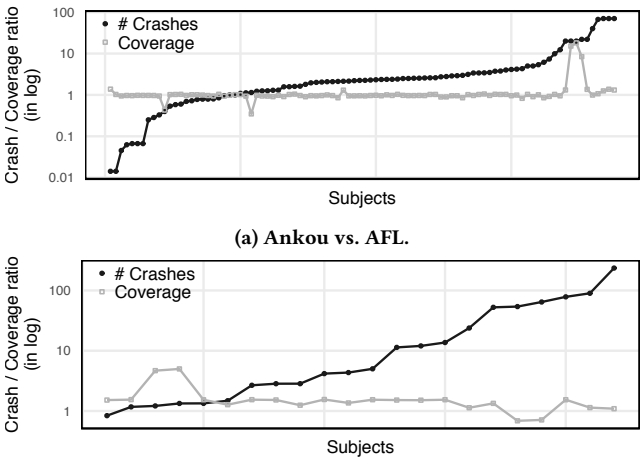
To answer the second question, we compared the test case generation speed of both Ankou and AFL. We chose AFL because it is a highly optimized fuzzer in terms of its fuzzing speed [59]. Figure 6 shows the test case generation throughput, which is a good measure for the cost of additional operations, as discussed in §6.1. We observed that Ankou was 35.0% slower than AFL on average, with 89% of the experiments being significant. However, this does *not* mean that Ankou is a worse fuzzer than AFL. Although Ankou is slow in generating test cases, it produces more meaningful ones, and thus, finds twice more bugs than AFL as we will see in §6.6.

Unexpectedly, Ankou showed a better throughput than AFL on 13% of the subjects. We thought this could be caused by Ankou achieving lower code coverage, making executions faster. However, the correlation between the coverage and the throughput ratios was only -0.6%. We believe Ankou found new regions of the programs that quickly terminate, while AFL did not. Overall, distance-based fuzzing significantly decreases the throughput, but it is worthwhile to perform more informed, hence more effective, seed pool updates.

6.6 RQ5: Comparison against Other Fuzzers

Although dynamic PCA is costly, it can enable higher software bug finding. To understand the practical impact of distance-based fuzzing, we answer the following two questions: (1) how effective Ankou is in terms of the number of unique crashes found? (2) how fast can Ankou find a crash?

6.6.1 Number of Crashes. We ran Ankou, AFL, and Angora on each subject. We then measured how many crashes were found for each subject along with the achieved branch coverage.



(a) Ankou vs. AFL.

(b) Ankou vs. Angora.

Figure 7: Comparison between Ankou and other fuzzers in terms of the number of crashes found and branch coverage.

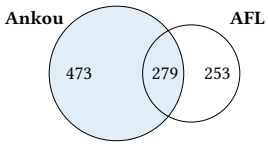
Figure 7a presents the result against AFL. In total, Ankou found 3K more unique crashes than AFL, which is 1.94× more on average. Ankou found more crashes on 75% of the subjects, on which 66% of the experiments where significant. On the other hand, the two fuzzers achieved more or less the same branch coverage: on average Ankou covered 1.27% more branches than AFL. Note that Ankou was able to find twice more crashes even though there was no big difference in terms of code coverage. This result indeed aligns with our key intuition: software bugs often manifest when we exercise a particular execution path, but not when we reach a node.

Figure 7b presents the result against Angora. Unlike AFL and Ankou, Angora requires DFSan [1] instrumentation to perform taint tracking, which makes it difficult to compile our benchmark. As a result, we were only able to compile about half of the packages. Among these, Angora found crashes in 22 subjects. Here, we report results only on those. On average, Ankou found 8.0× more crashes than Angora. Ankou prevailed on most subjects, and half of them showed strong statistical significance. These results confirm using the distance-based fitness function leads to better crash finding.

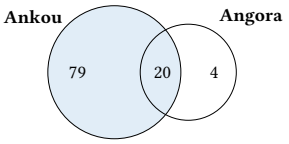
6.6.2 Time-To-Exposure of Crashes. We also measured how much time each fuzzer spends to find the first crash. On the subjects where both AFL and Ankou found crashes, Ankou was 27% faster in finding the first crash. Similarly, on the subjects where both Angora and Ankou found crashes, Ankou found them 68% earlier. This result also confirms the effectiveness of Ankou against state-of-the-art fuzzers in terms of its bug-finding ability.

6.7 Examination on Bugs Found

In §6.6 (RQ5), we reported the average number of crashes found for six repeated fuzzing experiments. During the whole experiment, Ankou found 93,754 crashes on the 150 subjects for 21,600 hours (= $24 \times 6 \times 150$). Although this number has its own value, we analyzed further to understand how many unique bugs each fuzzer found. This is important, as noted by Klees *et al.* [28], because multiple unique crashes may be due to the same bug.



(a) Ankou vs. AFL.



(b) Ankou vs. Angora.

Figure 8: Comparison of bugs found.

Table 2: Comparison between fuzzers by the number of unique bugs when triaged with stack hash.

	# of Bugs Found		# of Bugs Found	
Package Name	**Ankou**	**AFL**	**Ankou†**	**Angora**
binutils	11	36	11	4
bison	58	71	36	2
catdoc	0	15	0	0
cflow	21	18	13	1
clamav	0	0	0	0
dwarf	2	2	2	2
GraphicsMagick	22	27	21	11
jasper	42	37	0	0
libav	1	7	0	0
libexiv2	82	59	16	4
libgxps	5	5	0	0
liblouis	18	11	0	0
libming	84	60	0	0
libncurses	48	53	0	0
libraw	2	4	0	0
libsass	155	12	0	0
libtasn1	0	0	0	0
libtiff	3	2	0	0
libtorrent	0	0	0	0
mpg123	0	0	0	0
nasm	28	12	0	0
pspp	168	99	0	0
tcpdump	0	0	0	0
vim	2	2	0	0
Total	752	532	99	24

† As mentioned in §6.6.1, we were not able to compile Angora on all the packages. For fair comparison, we report bugs found by Ankou only on the subjects that Angora was able to run on.

Unfortunately, manual inspection was not an option as there were simply too many crashes. We originally tried to run ASan [49] to triage the crashes, but it failed to detect the root cause of many crashes. Therefore, we decided to use safe stack hash [12] instead, which works the same as the classic stack hash [41] with one exception: when there is an unreachable return address in the stack-trace, it stops traversing the stack. In our experiment, we computed the safe stack hash of the top five stack-trace entries of each crash. Although there are advanced crash triaging algorithms [17, 56], it is beyond the scope of this paper to adopt such techniques.

Figure 8 and Table 2 represents the number of unique bugs found after running the safe stack hash on all the crashes found. Overall, Ankou found 1.4× and 4.1× more unique bugs than AFL and Angora, respectively. There were overlaps, but there were a higher number of bugs that only Ankou was able to find. All these results confirm the practicality of Ankou in terms of bug finding.

7 DISCUSSION

First, we define the execution distance (Definition 3.2) as the Euclidean distance in the branch-hit-count space Ω_p. Although we

believe the choice of Euclidean distance is intuitive, one may consider a different distance metric such as Manhattan distance. Furthermore, the fitness function is defined as the minimum distance from a test case execution to the seed pool executions. While this is intuitively the amount of discovery made by this new test case, there may be a more optimal way of setting the fitness function. We see improving this area as promising future work.

In our experiment, the dynamic PCA was always able to reduce the state space with an acceptable information loss. However, there is no guarantee that it will be the case for all programs. We leave it as future work to prove a theoretical bound of its information loss.

With the adaptive seed pool update, the fitness threshold θ_{fit} is adaptively set to the minimum execution distance between any two seeds. However, there may be opportunities to choose a more appropriate threshold by not limiting ourselves to the contents of the seed pool. For example, refused test cases, even though they were not included in the pool, may be able to provide useful information to help this choice. Designing an optimal strategy for updating the seed pool is beyond the scope of this paper.

8 RELATED WORK

Fuzzing. Fuzzing has shown remarkable success in various areas [9, 11, 12, 23, 25, 26, 32, 33, 35, 37, 39, 43–47, 51, 57]. In the context of fuzzing, usage of the evolutionary algorithm was first introduced by Sidewinder in 2006 [19] and popularized by AFL and LibFuzzer [5, 58]. Ankou is also a grey-box fuzzer built upon the evolutionary framework. However, its uniqueness is its leverage of an informative fitness function that we call distance-based fitness, which deals with the considerably high dimensionality of the program state space, compared to the existing fitness functions.

Improving Fitness Function. There have been several research papers on improving the information given to, and the objective of the fitness function. CollAFL [22] improves information quality by avoiding hash collisions, thus indirectly enhances the fuzzer fitness function. Although it gains by avoiding imprecision, its fitness is still based on branch-hit-count states, so it suffers from the local optimum problem. PerfFuzz [31] leverages multi-dimensional feedback considering both code coverage and execution counts to tackle the local optimum problem. Eclipser [15] uses branch distances [40] to guide their search towards solving linear and monotonic constraints. Angora [14] augments its fitness function by considering the calling context when calculating branch coverage. However, none of these approaches handles the high-dimensionality problem of employing an informative fitness function. Our distance-based fitness function is complementary to them.

Distance between Test Cases. Feldt *et al.* [20] proposed a distance quantifying the difference between test cases. Unlike execution distance (see Definition 3.2), which is based on the execution of a program, this one is based on the input contents alone. It could still have been used in complement to $\delta_{\mathbf{B}}$ if it was not for its high computational cost. Pinilla-López *et al.* [36] compute PCA on the most recently discovered seeds to bias the seed scheduling. Although their work shares the same intuition in conceptualizing the state space, our approach differs both in goal as well as in the underlying technique. Our goal is guiding a fuzzing campaign using a fitness function, while theirs is modifying the seed scheduling algorithm. Moreover, scheduling algorithms can only be informed by seeds already chosen by their fitness function. However, our approach recognizes information from all the generated test cases.

Seed Scheduling. Starting from Woo *et al.* [55] seed scheduling has been a popular topic for improving fuzzers. AFLGo [10] and Hawekeye [13] combine fuzzing with information extracted from static analysis to direct fuzzers. AFLFast [11] suggests power scheduling, which assigns more energy to seeds that achieve higher code coverage. Cerebro [34] enhances seed scheduling based on a variety of objectives such as code complexity and code coverage of seeds. Such improvement has the benefit of focusing on a tiny subset of test cases already selected by the user or the fitness function, i.e., the seed pool. Unfortunately, we cannot directly apply these techniques to a fitness function due to its harsh performance requirement: it needs to run for every single test case.

Advanced PCA. Roweis [48] suggests an expectation maximization algorithm for computing PCA. It does not need the covariance matrix, and only calculates the desired number of principal components. However, this approach requires all the samples to be given at the beginning of the algorithm, which does not meet our needs since fuzzers generate samples throughout the fuzzing campaign. On the other hand, the online PCA [42, 54] aims to compute principal components on the fly: whenever new data is acquired, it updates the current principal components. This solution is not suitable for grey-box fuzzing as each of the online PCA updates has a complexity of $O(n^2)$, where n is the number of dimensions of the original space, i.e., the number of branches. Whereas the time complexity of dynamic PCA is linear in n. Other approaches such as random projection based online PCA [24] and stochastic PCA [8, 50] have a linear complexity. This is achieved by discarding much of the available data, unlike the dynamic PCA, which includes most of the data by updating the covariance matrix and its basis improvement mechanism. Furthermore, none of the approaches above includes a discount factor, described in §4.3.

9 CONCLUSION

We designed and implemented Ankou, the first grey-box fuzzer that operates with a high dimensionality representation of the program state space. Ankou employs distance-based fitness function, which provides too much information about program executions to consume in practice. However, we transform the information obtained by the fitness function with our novel dimensionality reduction technique that we refer to as dynamic PCA. As a result, we were able to greatly improve the current state of grey-box fuzzing in terms of its bug finding ability. We made both our source code and benchmark public to support open science.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their constructive feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697, Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security).

REFERENCES

[1] [n.d.]. Data Flow Sanitizer. http://clang.llvm.org/docs/DataFlowSanitizer.html.
[2] [n.d.]. Fidgety AFL. https://groups.google.com/forum/#!topic/afl-users/fOPeb62FZUg.
[3] [n.d.]. The Go Programming Language. https://golang.org.
[4] [n.d.]. Gonum Numeric Library. https://www.gonum.org.
[5] [n.d.]. LibFuzzer. http://llvm.org/docs/LibFuzzer.html.
[6] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. Google Testing Blog.
[7] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. 1–10.
[8] Raman Arora, Andy Cotter, and Nati Srebro. 2013. Stochastic optimization of PCA with capped MSG. In *Advances in Neural Information Processing Systems*. 1815–1823.
[9] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the Network and Distributed System Security Symposium*.
[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2329–2344.
[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1032–1043.
[12] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 725–741.
[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2095–2108.
[14] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the IEEE Symposium on Security and Privacy*. 855–869.
[15] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*. 736–747.
[16] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the International Conference on Very Large Data Bases*. 426–435.
[17] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the International Conference on Software Engineering*. 820–831.
[18] Al Danial. [n.d.]. Count Lines of Code: Coverage Tool. http://cloc.sourceforge.net/.
[19] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. 2006. "Sidewinder": An Evolutionary Guidance System for Malicious Input Crafting. In *Proceedings of the Black Hat USA*.
[20] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. 223–233.
[21] John GF Francis. 1961. The QR transformation a unitary analogue to the LR transformation. *Comput. J.* 4, 3 (1961), 265–271.
[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 660–677.
[23] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the International Conference on Automated Software Engineering*. 50–59.
[24] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. 2011. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM review* 53, 2 (2011), 217–288.
[25] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2345–2358.
[26] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the Network and Distributed System Security Symposium*.
[27] Ian T. Jolliffe. 2011. *Principal Component Analysis*. Springer.
[28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2123–2138.
[29] lafintel. 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.
[30] Joel Lehman and Kenneth O Stanley. 2008. Exploiting Open-Endedness to Solve Problems through the Search for Novelty. In *Proceedings of the International Conference on Artificial Life*. 329–336.
[31] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the International Symposium on Software Testing and Analysis*. 254–265.
[32] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the International Conference on Automated Software Engineering*. 475–485.
[33] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 627–637.
[34] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 533–544.
[35] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 521–532.
[36] Jorge Pinilla López. 2019. Improving fuzzing performance using hardware-accelerated hashing and PCA guidance. https://cs.anu.edu.au/courses/csprojects/19S1/reports/u6759601_report.pdf.
[37] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2946563
[38] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou. https://github.com/SoftSec-KAIST/Ankou.
[39] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed Fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 548–560.
[40] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*. 153–163.
[41] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the USENIX Security Symposium*. 67–82.
[42] Jiazhong Nie, Wojciech Kotłowski, and Manfred K. Warmuth. 2013. Online PCA with Optimal Regrets. In *Proceedings of the International Conference on Algorithmic Learning Theory*. 98–112.
[43] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the USENIX Security Symposium*. 729–743.
[44] Jibesh Patra and Michael Pradel. 2016. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data*. Technical Report TUD-CS-2016-14664. TU Darmstadt.
[45] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681
[46] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*.
[47] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the USENIX Security Symposium*. 861–875.
[48] Sam Roweis. 1997. EM Algorithms for PCA and SPCA. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems*. 626 – 632.
[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*. 309–318.
[50] Ohad Shamir. 2015. A stochastic PCA and SVD algorithm with an exponential convergence rate. In *International Conference on Machine Learning*. 144–152.
[51] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry Practice of Coverage-Guided Enterprise Linux Kernel Fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 986–995.
[52] Gilbert Strang. 2003. *Introduction to Linear Algebra* (3 ed.). Wellesley-Cambridge Press.
[53] Charles F Van Loan and Gene H Golub. 1983. *Matrix computations*. Johns Hopkins University Press.
[54] Manfred K. Warmuth and Dima Kuzmin. 2008. Randomized Online PCA Algorithms with Regret Bounds that are Logarithmic in the Dimension. *Journal of Machine Learning Research* 9 (2008), 2287–2320.
[55] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 511–522.
[56] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump. In *Proceedings of the ACM Conference on Computer and Communications Security*. 529–540.

[57] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing Without Valid Seed Inputs. In *Proceedings of the International Conference on Software Engineering*. 712–723.

[58] Michal Zalewski. [n.d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[59] Michal Zalewski. [n.d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.