Effective Fuzzing within CI/CD Pipelines (Registered Report)

Arindam Sharma
Imperial College London
United Kingdom
arindam.sharma@imperial.ac.uk

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

Jonathan Metzman
Google
USA
metzman@google.com

Abstract

Deploying fuzzing within CI/CD pipelines can help ensure safe and secure code evolution. Directed greybox fuzzing techniques such as AFLGo are a good match for the CI/CD context. These techniques prioritise inputs based on estimated distances to the changed code. Unfortunately, computing these distances is often expensive, making the techniques impractical for short CI/CD runs.

In this paper, we propose an AFLGo-based technique called PaZZER, which optimises the distance calculation by dropping the expensive control-flow graph component and computing the call-graph component in an incremental fashion. Preliminary results are promising, showing that PaZZER can make CI/CD testing feasible for large applications: e.g., for Objdump the distance computation time is decreased from 34 min to just 2.5 min, with a further 2.3 min saved when an incremental algorithm is used. The significant time reduction in distance computation allows PaZZER to use most of the time on actual fuzzing, making it practical for short CI/CD runs of around 10 minutes.

Our planned full evaluation will involve real-world commits from a diverse set of nine applications of different sizes. This will include coverage experiments and an ablation study to investigate the impact of PaZZER's design decisions, and a bug-finding case study comparing it against AFLGo and Google's CIFuzz. We will assess the benefits and effectiveness of our approach in terms of patch coverage, patch proximity, distance computation time, and time-to-exposure for bugs.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Directed greybox fuzzing, CI/CD testing, AFLGo

ACM Reference Format:
Arindam Sharma, Cristian Cadar, and Jonathan Metzman. 2024. Effective Fuzzing within CI/CD Pipelines (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24), September 16, 2024, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3678722.3685534

FUZZING '24, September 16, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1112-1/24/09
https://doi.org/10.1145/3678722.3685534

1 Introduction

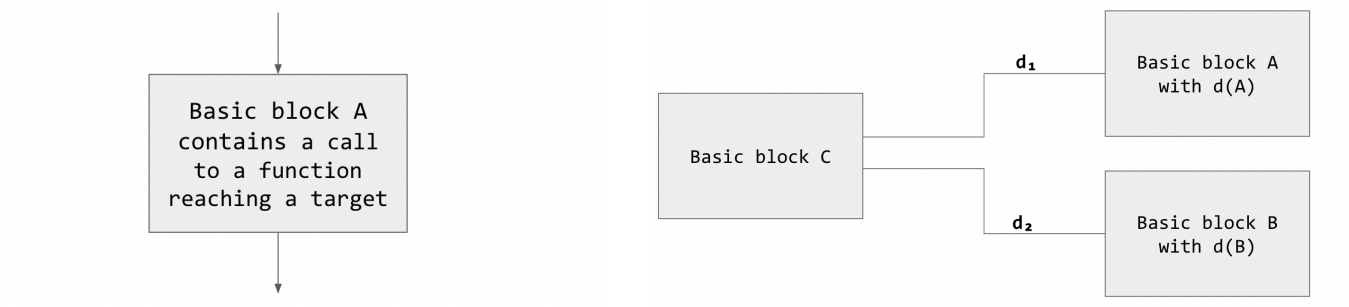
Evolution is an inevitable characteristic of software, but the high pace at which modern software evolves poses a key threat to its reliability. The most popular and effective way of ensuring safe software evolution is through the use of Continuous Integration (and Continuous Development), CI/CD pipelines. Typically, these pipelines run a developer-written test suite on each commit, to check that the introduced changes did not introduce any issues. However, with software becoming more complex, these manual tests may easily miss corner cases responsible for bugs and security vulnerabilities.

A promising direction is the use of greybox fuzzing within the CI/CD pipelines to supplement developer-written test suites. In particular, *directed greybox fuzzing* adapts greybox fuzzing to target specific parts of the code and has gathered significant attention recently. The pioneering work in this area is AFLGo [3], which extends the popular AFL fuzzer [26] with a heuristic based on the distance to the target code. Several subsequent techniques have aimed to improve various aspects of AFLGo. For example, Du et al. [11] extend AFLGo by taking into account data dependencies for distance computation, while Wüstholtz et al. [24] provide an on-demand, online static analysis for distance estimation. WindRanger [6] builds on top of AFLGo by considering basic blocks that have a better chance of reaching the target(s), while Liang et al. [15] extend AFLGo to better deal with multiple targets. WAFLGo [25] follows the same trend and extends AFLGo by adding support for multiple targets and by directing fuzzing efforts towards sensitive regions following a software patch.

Existing research from Klooster et al. [12] has shown that in general, employing per-commit fuzzing campaigns of 10 min strikes a good balance between the desires of developers for quick feedback and the effectiveness of fuzzing. Additionally, campaigns of 10 min can be just as effective as ones that take 8 hours, especially if lengthier campaigns are still regularly used to fuzz snapshots of the repository.

Unfortunately, while AFLGo is designed to target changes, for many larger applications it is unable to achieve good results within a short per-commit time budget. The key bottleneck consists of time spent by the tool in computing the necessary distances to the lines of code changed by the patch under analysis before the actual fuzzing campaign starts. Not counting the distance calculation and other pre-computation steps in the fuzzing budget [3, 6, 11, 15, 27] skews the perceived efficiency of the fuzzer and does not reflect a real-world deployment, where the pre-computation stage needs to run whenever a new code change arrives.

In this work, we address this scalability challenge via two interrelated optimisations. First, we show that one of the distance types computed by AFLGo can be dropped, without a significant degradation in its effectiveness. Second, we replace its distance computation



(a) Basic blocks from which the target is directly reachable. **(b) Basic blocks that can indirectly reach the target.**

Figure 1: Types of basic blocks used in AFLGo's basic-block level, CFG-based distance computation.

with an incremental algorithm, and demonstrate the advantages and trade-offs incurred by various such algorithms.

The rest of this registered report is structured as follows. §2 provides some background information on AFLGo and a motivating case study. Then, §3 discusses the optimisations proposed by our system called PAZZER.[1] §4 presents out preliminary results and §5 the planned evaluation for the full registered paper submission. Finally, §6 discusses related work and §7 concludes.

2 Motivating Case Study

In this section we briefly describe the inner workings on AFLGo [3] (§2.1) and present a motivating case study that highlights the time spent by AFLGo in different stages.

2.1 Background for AFLGo

To effectively generate inputs that reach a set of targets (typically the targets consists of the changed parts of the code), AFLGo estimates the distance from each input in the fuzzing queue to the targets. The inputs that are *closest* to a *target* are then prioritised.

To compute the distance from an input to a set of targets, AFLGo computes the distance from each basic block traversed by the input to the set of targets, and then returns the average of these distances. The distance from a basic block to a target is pre-computed before the actual fuzzing campaign begins.

Each pre-computed distance has two components: a *function-level target distance* and a *basic-block-level target distance*. The former computes the distance between the function where the basic block resides and the functions where the targets reside, while the latter computes the distance between the basic block and all other basic blocks that call a target function.

The function-level target distance is based on the Call-Graph (CG) extracted from the system under test (SUT) and is hence *interprocedural*. Essentially, it computes the distance between a function n and target functions T_f and is defined as the harmonic mean of the distances (computed using DIJKSTRA's algorithm on the call graph) between n and the target functions T_f:

$$d_f(n, T_f) = \begin{cases} \text{undefined} & \text{if } R(n, T_f) = \emptyset \\ \left[\sum_{t_f \in R(n,T_f)} d_f(n, t_f)^{-1}\right]^{-1} & \text{otherwise} \end{cases} \quad (1)$$

with $R(n, T_f)$ being the set of all target functions that can be reached from n.

The *basic-block-level target distance* is *intraprocedural* and uses the Control Flow Graphs (CFGs) of the functions in the SUT. Figure 1 shows the two different kinds of basic blocks that are used to compute this distance. More exactly, Figure 1a shows the kind of basic blocks from where the target is directly reachable. For such a basic block A the distance estimate is

$$d(A) = c \cdot \min\{\text{CG-based distance to target function}\} \quad (2)$$

with c a configurable constant (with default value 10 in AFLGo).

The second kind of basic blocks, as shown in Figure 1b, are the ones from which the target is indirectly reachable. For a basic block C, the distance is computed as,

$$\begin{aligned} d(C) &= \text{harmonic-mean}(d_1 + d(A), d_2 + d(B)) \\ &= \left((d_1 + d(A))^{-1} + (d_2 + d(B))^{-1}\right)^{-1} \end{aligned} \quad (3)$$

In the rest of the paper, for brevity, we refer to the first kind of distance as *CG* (Call Graph) distance, and to the second one as *CFG* (Control-Flow Graph) distance.

2.2 Performance Analysis for AFLGo

In order to get a better understanding of how suitable AFLGo is in the context of CI/CD fuzzing, we run the tool on four benchmarks which were also part of AFLGo's evaluation [3]. The benchmarks are picked to be of varying sizes, representing very small (GIFLIB [1]), small (JASPER [10]), medium (LIBMING [2]) and large (OBJDUMP, part of BINUTILS [7]) benchmarks. We run our experiments on machines with Intel Core i3-8100 CPU @ 3.60GHz with 32GB of memory, running UBUNTU LINUX 20.04 LTS x86_64.

Table 1 shows how much time is spent computing the two types of distances, together with their sum. As can be seen, the distance computation step takes significant time in the context of short CI/CD runs.

[1] The name is a combination of PAtch and fuZZER.

Table 1: Time taken in distance computation for each benchmark. Benchmark versions and sizes in lines of code (LOC) are also provided. Objdump is part of Binutils and the size is our best estimate taking its dependencies into account.

Benchmark	Version	LOC	Distance comput. (s)		
			CG	CFG	Total
Giflib	72e31ff	7,259	1	32	33
Jasper	142245b	28,256	2	181	183
Libming	b72cc2f	82,303	2	192	194
Objdump	a6c21d4	570,040	154	1,866	2,020

For Giflib, Jasper, and Libming, the total distance computation step is within the fuzzing time budget of 600 s (10 min). However, in the case of Jasper and Libming the CFG distance already takes substantial time, at over three minutes.

For Objdump, representative of large benchmarks, the distance computation step takes 2,020 s (34 min), which makes the technique unsuitable for use within CI/CD pipelines. This is dominated by the CFG computation, which takes 31 min.

These findings motivate our proposed approach of dropping the CFG distance computation and improving the re-computation of the CG distance. In particular, we aim to understand the impact of dropping the CFG distance component on the capabilities of AFLGo for short CI/CD fuzzing runs, and the time savings achieved by incrementally computing CG distances.

3 PaZZER

PaZZER updates the AFLGo algorithm in two main ways: it drops the CFG distance computation (§3.1) and makes the CG distance computation incremental (§3.2).

3.1 CG Distance Computation

For a basic block b within a function f, the distance $d(b,t)$ to a target node t is directly given by the precomputed function-level distance D_f between b_f, the function where b resides, and t_f, the function where t resides.

$$d(b,t) = D_f(b_f, t_f)$$

In PaZZER, D_f is computed using an incremental algorithm:

$$D_f(b_f, t_f) = \text{IncrementalAlgorithm}(b_f, t_f, D_f^{\text{old}}, \text{CG}_{\text{Updates}})$$

where IncrementalAlgorithm represents the chosen incremental algorithm, D_f^{old} represent the set of distances computed in the old version of the call graph, and CG_{Updates} are the updates to the call graph done in the new version.

3.2 Incremental Algorithms

The incremental algorithm used by PaZZER can have an important impact on its performance and effectiveness. The literature offers a wide choice of incremental shortest path algorithms, each presenting different tradeoffs in terms of optimality (whether it computes shortest paths or an approximation), performance, and memory consumption. We plan to investigate algorithms representative of these choices, which we summarise in Table 2.

We have selected a total of three incremental algorithms: LPA*, Anytime D*, and HPA*. We have picked one algorithm to represent each of the three classes of algorithms based on optimality: optimal, sub-optimal and near-optimal. We briefly describe each in turn.

LPA* is representative of an optimal but moderately efficient shortest-path algorithms. Optimal algorithms guarantee a shortest-path solution, but they are more expensive than sub-optimal and near-optimal algorithms. LPA* is based on the A* algorithm while extending it with incremental capabilities. It guarantees the optimality of the shortest path solution, ensuring accuracy and precision in distance computation. LPA* algorithm is useful in cases where the graph evolves slowly and less drastically. In the context of this work, it would be useful for SUTs where the changes do not alter the underlying call graph too much by adding/removing functions and calls.

Anytime D* is representative of highly efficient algorithms that produce a sub-optimal solution that gets refined over time. Consequently Anytime D* is categorised as a sub-optimal and moderately efficient algorithm. While it does not always guarantee the shortest path, it provides a solution quickly and can improve the solution quality over time, making it suitable for scenarios where time constraints are critical. The ability to generate a solution very quickly might be critical for our larger benchmarks. The memory footprint of this algorithm can be quite high when handling significant changes as more of the nodes need reprocessing.

HPA* is representative of a fast algorithm that produces "near-optimal" solutions for certain target nodes, useful for cases where a certain part of the codebase changes a lot. HPA* is considered near-optimal since it balances between optimality and efficiency by using hierarchical abstraction to reduce the complexity of the path-finding problem, thus offering a solution that is close to optimal in a more computationally efficient manner. The hierarchical abstraction allows it to scale well for large call graphs, but the initial setup is complex and the initial memory consumption is high due to the fact that it needs to maintain the abstraction in memory.

With these algorithms in place, our aim is to get a better understanding of the optimal use cases for each of the three algorithms. Given that a key challenge for the deployment of a fuzzer within a CI/CD pipeline is the highly evolving nature of the system, it will be important for PaZZER to be able to select the best fitting incremental algorithm. We hypothesise that larger systems that evolve quite rapidly would benefit from highly efficient but sub- or near- optimal algorithms. However, for smaller systems where a less efficient but optimal algorithm is feasible, that would be the better choice.

3.3 Directed Fuzzing

PaZZER is built on top of the popular AFLGo fuzzer [3]. While several different directed fuzzers have been proposed in recent work, our decision to build on top of AFLGo stems from the fact that it provides a simple, yet effective solution to the problem. Furthermore, it offers a stable implementation, which has been used and extended successfully by several projects.

Table 2: Summary of incremental shortest-path algorithms we plan to use in PaZZER, based on their use cases.

Algorithm	Optimality	Description	Ideal Use Case
LPA* [14]	Optimal	Combination of DynamicSWSF-FP [20] and A*. Uses information from previous runs to reduce the number of nodes examined.	Managing call graphs in dynamic codebases where call relationships change occasionally and those changes are largely local, maintaining efficient path updates.
Anytime D* [16]	Sub-Optimal	Variant of D* Lite [13] combined with Anytime Repairing A*. Finds suboptimal paths quickly and improves with more time given.	Quickly identifying distance estimates in very large and dense call graphs. For example, in case of large repositories that undergo frequent updates, Anytime D* can rapidly find initial distances after changes and refine them as more time becomes available.
HPA* [9]	Near-Optimal	Hierarchical Path-Finding A* for pathfinding in large graphs with many units. Breaks the graph hierarchically for near-optimal paths.	Ideal for large, complex call graphs in a monolithic codebase where dealing with the entire Call Graph might be infeasible and hence breaking the graph into smaller sections (modules) improves the efficiency of pathfinding and analysis.

4 Preliminary Results

This section presents our preliminary results, which indicate that PaZZER has the potential of making directed greybox fuzzing, and AFLGo in particular, feasible in a CI/CD context.

We first investigate the impact of dropping the expensive CFG distance component in §4.1, and then measure the impact of an incremental approach to CG distance computation in §4.2. We run the experiments on the same machines described in §2.2.

4.1 Impact of Dropping CFG Distances

To understand the impact of dropping CFG distances and using only CG distances on the effectiveness and performance of AFLGo, we have run an experiment measuring the time needed to find known crashes in the four benchmarks introduced in §2.2 (Jasper, Libming, Giflib and Objdump), with and without CFG distance computation.

We reuse here the experimental setup from the AFLGo paper [3]. In particular, we mark as targets the method calls from the stack trace associated with each crash. For our crashes, we select from the ones provided by the AFLGo artifact:[2] for Jasper, Libming and Objdump, we use CVEs, while for Giflib a non-exploitable crash bug as no CVE is provided. For each crash, we measure the Time-to-Exposure (TTE) for both AFLGo and PaZZER, i.e. the time required to generate an input that triggers the crash. Note that for this experiment, the incremental component of PaZZER is not used. We repeat our experiments 20 times to gain statistical confidence in our results.

Table 3 presents our results. For both AFLGo and PaZZER, we measure the total time it takes to find the crash in each SUT. We further break down this time into the distance computation and the fuzzing run components.

[2]From https://github.com/aflgo/aflgo/tree/master/examples

Our results show that PaZZER finds all the crashes in significantly less time that AFLGo, with a speedup between 1.1 x to 5.2 x compared to AFLGo. This gain is especially crucial for larger benchmarks like Objdump where the distance computation time for AFLGo far exceeds the 10 min fuzzing budget. In such cases, the use of PaZZER makes it feasible for such benchmarks to be able to utilise directed fuzzing in a CI/CD setting.

With these preliminary results, we can get a reasonable amount of confidence in this particular design element of PaZZER. More precisely, we can see that by only using a CG-based distance measure in PaZZER, not only can we avoid expensive computation steps and make the process more scalable, but also do not adversely impact the effectivess of the directed fuzzing.

4.2 Impact of Incrementally Computing CG Distances

The second part of our preliminary evaluation looks at understanding the impact of incrementally computing CG distances starting from the distances already computed for the previous version of the SUT. In particular, we report the performance of the incremental LPA* algorithm against the non-incremental Dijkstra's algorithm (which is the non-incremental algorithm used by AFLGo) on programs that undergo changes over time.

In complex changing call graphs, recomputing path distances with Dijkstra's algorithm can be inefficient and resource-intensive. Dijkstra's algorithm requires recomputation of the entire path from the start node to the goal node whenever there is a change in the graph, leading to significant computational overhead. As call graphs often do not change that much, there are important missed opportunities in terms of reusing previously computed path distances.

The Lifelong Planning A* (LPA*) algorithm incrementally updates shortest path information when changes occur in the graph.

Table 3: Time-to-Exposure (TTE) comparison between AFLGo and PaZZER.

Benchmark	CVE/Version	AFLGo-TTE (s)			PaZZER-TTE (s)			Speedup
		Distance	Fuzzing	Total	Distance	Fuzzing	Total	
Giflib	72e31ff	33	89	122	1	108	109	1.1 x
Jasper	CVE-2015-5221	183	62	245	2	66	68	3.6 x
Libming	CVE-2018-8962	194	176	370	2	192	194	1.9 x
Objdump	CVE-2017-8392	2,020	243	2,263	154	284	438	5.2 x

Table 4: Performance comparison of PaZZER (with LPA* algorithm) and AFLGo (with Dijkstra's algorithm).

Benchmark	Call Graph		Time (s)	
	Nodes	Edges	AFLGo	PaZZER
Giflib	65	119	1	1
Jasper	744	1,694	2	2
Libming	434	1,347	2	2
Objdump	21,596	51,318	154	14

As discussed in §3.2, it is an example of an optimal incremental algorithm of moderate efficiency. By incrementally updating its search for the optimal path based on changes in the graph, LPA* minimises computational costs and avoids redundant exploration.

Experimental setup. To evaluate the impact of incrementality in the context of CI/CD fuzzing, we have implemented LPA* in AFLGo. The starting point for our implementation is the Dijkstra's algorithm implementation from Python's NetworkX [8] package, which is the same implementation used by AFLGo. The implementation is designed to handle unweighted graphs, as needed in the context of call graphs.

In order to simulate a bug-inducing commit scenario, we take our running benchmarks from Table 1 and add a bug-injecting commit. In particular, we select a function in each benchmark, and add a function call to another selected function. We further add an `abort()` statement at the beginning of the called function to induce a crash. This changes the call graph by adding an extra edge to it.

We then run both the non-incremental Dijkstra's algorithm and the incremental LPA* algorithm and measure the amount of time it takes for each. Table 4 presents our findings.

For the small and medium-sized benchmarks (Giflib, Jasper and Libming), due to the relatively small graph sizes, the call graph distance computation times are small, 1-2s, so the incremental algorithm does not make a difference in practice.

However, for the large Objdump benchmark, with a call graph with 21,596 nodes and 51,318 edges, the incremental aspect of the LPA* algorithm results in a time saving of 140 s. While not as significant as in the case of dropping CFG distance computation, this amount of saving is nevertheless important when a short CI/CD fuzzing run of 10 min is considered.

5 Planned Evaluation

In this section, we discuss the evaluation that we plan to perform for the full registered paper submission. We start by discussing our research questions (§5.1) and the systems under test (§5.2). We then present two experiments: one focusing on coverage and an ablation study (§5.3) and the other focusing on bug-finding (§5.4). For each experiment, we present the techniques being compared, the evaluation metrics, the methodology and the experimental settings.

5.1 Research Questions

Our research questions (RQs) are as follows:

RQ1: What is the impact of dropping the basic-block level target distance used by AFLGo on the capabilities of directed fuzzing for short fuzzing runs?

RQ2: How does the incremental computation of call-graph-based distances perform in comparison to recomputation-based non-incremental methods in terms of accuracy and efficiency?

RQ3: What are the trade-offs between computation time and fuzzing capabilities when using different incremental algorithms?

RQ4: How does the proposed incremental, directed fuzzing infrastructure scale with the size and complexity of codebases, considering typical CI/CD constraints such as time limits and resource availability?

5.2 Systems under Test

To comprehensively evaluate our proposed technique, we have selected a diverse set of systems under test (SUTs) from the OSS-Fuzz project [21], summarised in Table 5. These SUTs encompass a diverse array of software projects, covering domains such as image handling (e.g., Giflib and Libjpeg-turbo), data processing (e.g., Jasper and Objdump), and secure communication (e.g., Libbpf and OpenSSL)

The chosen SUTs are categorised into small, medium, and large (three of each), based on the number of lines of code, as reported by OSS-Fuzz. This categorisation aligns with the proposed research questions, allowing us to analyse the performance and efficacy of our technique across SUTs of different sizes. Small-scale SUTs—Giflib, Jasper and Libbpf—provide insights into the effectiveness of the technique in detecting vulnerabilities in compact codebases. Medium-scale SUTs—Libming, Selinux and Libjpeg-turbo—allow us to evaluate our technique's efficacy in moderately-sized projects. Finally, large-scale SUTs—OpenSSL, Systemd and Objdump—enable

Table 5: The benchmarks from OSS-Fuzz [21] that we plan to use for the full evaluation.

Name	Scale	Lines of Code	CIFuzz Integration	Notes
Giflib	Small	7,259	✗	Library for handling GIF files.
Jasper	Small	28,256	✗	C library for parsing and processing JSON data.
Libbpf	Small	36,670	✓	A C library for loading, interacting with, and managing BPF (Berkeley Packet Filter) programs in the Linux kernel.
Selinux	Medium	63,634	✓	A security architecture integrated into the Linux kernel that provides mandatory access controls.
Libjpeg-turbo	Medium	68,397	✓	A JPEG image codec that uses SIMD instructions to accelerate baseline JPEG compression and decompression.
Libming	Medium	82,303	✗	Library for generating SWF ("Flash") format movies.
Systemd	Large	280,393	✓	A system and service manager for Linux operating systems that provides parallelized booting, on-demand starting of daemons, and more.
Objdump	Large	570,040	✗	Library for displaying various information about object files, including disassembly and binary content.
OpenSSL	Large	820,725	✓	Widely-used cryptographic library.

us to assess the scalability and robustness of our approach in handling complex, extensive codebases.

5.3 Coverage Experiments and Ablation Study

The first set of experiments will focus on understanding the impact of different design decisions in PaZZER on an unbiased selection of commits from each SUT. These experiments will focus on coverage as our primary metric.

Techniques under evaluation. Our evaluation will compare the following tools and configurations:

(1) **AFLGo**: Representative of a general and effective form of directed fuzzing, along with its complete distance computation functionality.
(2) **PaZZER-No-Increment**: PaZZER without incremental distance computation but with AFLGo's expensive CFG distance computation dropped.
(3) **PaZZER-LPA***: PaZZER with LPA* as its incremental algorithm.
(4) **PaZZER-Anytime-D***: PaZZER with Anytime D* as its incremental algorithm.
(5) **PaZZER-HPA***: PaZZER with HPA* as its incremental algorithm.

Evaluation metrics. In this part of the evaluation, we will focus on the following metrics:

- **Patch coverage.** For each code change (patch) considered in each SUT, we will measure the total patch coverage achieved by the inputs produced by each tool. For patches with a single target (i.e. a single basic block), this is either 0% or 100% and indicates whether or not the input set was able to reach the patch. For patches consisting of multiple target basic blocks, this metric is defined as the fraction of basic blocks reached by executing the input set. This will provide an insight into the impact of the various choices we made regarding the distance metric and the incremental algorithms on the patch coverage capabilities of the directed fuzzer.
- **Patch proximity.** If a tool does not manage to generate inputs that reach the patch, we will measure *patch proximity*, i.e. how close the generated inputs got to the patch. We will use a call-graph-based distance to the target to compute patch proximity, and in the case of multiple targets, we will pick the input with the smallest average distance.
- **Time saved in distance computation**. Between incremental configurations of PaZZER and the non-incremental AFLGo, we will measure the amount of time spent in distance computation. This will give us the average amount of time saved by making the directed fuzzing infrastructure use an incremental CG distance computation.

Methodology and experimental settings. We will run the five tools on a set of 27 commits from each of our nine SUTs. Each tool will be given a total fuzzing budget of 10 min, which includes the time required to run the distance computation step. The rationale behind this time budget is two-fold: (1) Since these tools need to run within a CI/CD pipeline on every single commit, a short response time is required by developers; and (2) Prior work has shown that a short 10 min run is often effective [12].

To address the impact of the size and diversity of changes on the performance of PaZZER, we introduce the following commit selection methodology:

(1) For each SUT, we pick all possible continuous sequences of three commits, starting with a commit from roughly five years ago. For each such sequence, we compute the average *commit size* and the average *commit spread*, where these metrics are defined as such:
   - *Commit Size:* The number of code lines added, modified, or deleted by the commit.
   - *Commit Spread:* The average number of unique source code files affected.
(2) We sort these sequences into two lists, the first based on average commit size and the second based on average commit spread.
(3) We then divide both these sorted lists into three parts corresponding to high, medium and low average values.
(4) From these sorted lists, we select a total of nine commit sequences, each containing three commits. These nine commit sequences correspond to the nine different combinations that can be made with the three (high, medium, low) classes for the two (average commit size, average commit spread) categories.

Given the commit selection timeframe of five years, we should have sufficient sequences for each combination, however for cases where this is not possible, we will skip that combination.

As expected, for each commit in a given sequence, we will consider the lines affected by the commit, and use these as targets for our five tools. Each tool is run on each commit for 10 min while saving all the inputs being generated during these runs. For AFLGo and PaZZER-No-Increment where there is no incremental component, we will simply re-run them each time for those 27 commits. For the remaining tools which have an incremental component to them, we will start from the distances computed for the previous commit. In particular, this will involve computing the distances for the commit preceding the given sequence (outside the 10 min time budget), under the assumption that those distances had already been computed when fuzzing the previous commit.

For all tools, we will also keep track of the amount of time spent (re)computing distances. Having a selection of commit sequences based on commit size and spread for each SUT, in conjunction with the mentioned metrics will allow us to understand the impact of incremental algorithms and their various variants on the efficiency of directed fuzzing.

For each run, we will compute the metrics discussed above, which in turn will be used to answer the RQs introduced in §5.1.

These experiments will be conducted on a cluster of multicore Linux workstations. The detailed specifications of the machines will be provided when presenting our full results.

Our planned experiments will require (excluding the time needed to compute the distances for the commit preceding each sequence):

$$\begin{aligned} &9 \text{ benchmarks} \times \\ &27 \text{ commits per benchmark} \times \\ &10 \text{ min per commit} \times \\ &20 \text{ repetitions} \times \\ &5 \text{ tools} \\ =\ &4{,}050 \text{ hours of CPU time} \end{aligned}$$

Performing 20 repetitions allows us to present average numbers for our given metrics with statistical confidence.

5.4 Bug-Finding Experiments

For our second set of experiments, we plan to focus on the bug-finding capabilities of PaZZER. We plan to extract several recent bugs reported in OSS-Fuzz for our SUTs, and understand whether PaZZER would have been able to find the bug within a short CI budget. We will also compare PaZZER with two other systems: AFLGo and CIFuzz.

CIFuzz [23] is a system offered by Google for running fuzz targets on pull requests. It is designed for SUTs integrated with OSS-Fuzz and hosted on GitHub. CIFuzz offers a very basic incremental feature, in which only the fuzz targets which have previously reached the files involved in the code change (pull request) are run.

Bug selection methodology. We plan to select 20 bugs reported by OSS-Fuzz in the SUTs of Table 5, restricting ourselves to those SUTs also integrated with CIFuzz. From our nine SUTs, five are also integrated with CIFuzz: Libbpf, Selinux, Libjpeg-turbo, Systemd and OpenSSL. We will select four bugs from each of these five benchmarks, for a total for 20 bugs.

Techniques under evaluation. Our evaluation will compare the following tools:

- **AFLGo**: The standard configuration of AFLGo. The distance computation step will be included in its time budget.
- **CIFuzz** The standard configuration of CIFuzz.
- **PaZZER** We will configure PaZZER with the incremental algorithm that performed best in the experiments of §5.3.

Evaluation metric. For each bug-introducing commit, we will record the amount of time it takes for each tool to generate the first input that exposes the bug, i.e. the Time-to-Exposure (TTE).

Methodology and experimental settings. We run each tool for 10 min, as for the experiments in §5.3. Since CIFuzz and PaZZER have incremental features, we first simulate the fact that they ran on the versions preceding the bug-introducing commit under analysis. We do so in two steps: first, we run all the fuzz targets in CIFuzz on the previous commit, for 10 min each, and gather coverage information. This coverage information will be used during the actual run on the error-introducing commits to use only the fuzz targets affected by the commit, as CIFuzz does. Second, we compute the CG distances using the previous commit. Of course, we do not count these steps as part of the 10 min time budget, as the assumption is that the fuzzers have already been run on previous commits.

Like before, these experiments will be conducted on a cluster of multicore Linux workstations. Our planned experiments will require (excluding the time needed to simulate the fact that the tool had run on the previous commit):

$$\begin{aligned} &20 \text{ bugs} \times \\ &10 \text{ min per bug} \times \\ &20 \text{ repetitions} \times \\ &3 \text{ tools} \\ =\ &200 \text{ hours of CPU time} \end{aligned}$$

The results of these experiments will provide important insights into: (1) whether bugs which take a large number of CPU hours to find with OSS-Fuzz could be found in less than 10 min with PaZZER; (2) whether the incremental aspect of PaZZER enables it to find bugs which are out of reach for AFLGo; and (3) how directed fuzzing compares in practice with the simple undirected CIFuzz deployment.

6 Related Work

Directed greybox fuzzing has gained significant attention due to its ability to focus fuzzing efforts on specific parts of the codebase, which is particularly beneficial for identifying vulnerabilities in critical or recently modified sections of software. The pioneering work in this area is AFLGo [3], on which PaZZER is based. AFLGo extends the widely-used AFL fuzzer [26] by incorporating a heuristic based on the distance to the target code. This approach guides the fuzzer to generate inputs that are more likely to exercise the specified parts of the code, enhancing the efficiency of the fuzzing process. Given AFLGo's effectiveness and popularity, it has served as the basis of a number of extensions. Some works modify various features of the AFLGo algorithm, while others present complementary ways to enhance the algorithm.

DAFL [11] extends AFLGo by incorporating data dependencies into the distance computation step. WindRanger [6] refines the target selection process by prioritising basic blocks that have a higher likelihood of reaching the specified targets, thus improving the efficiency of the fuzzing process. Addressing the issue of multiple targets affecting the directed fuzzer, Liang et al. [15] enhance AFLGo to effectively handle scenarios with multiple fuzzing targets by using a custom distance metric, optimizing the fuzzing process across different parts of the codebase. Similarly, FishFuzz [27] employs a hierarchical scheduling strategy to efficiently manage and prioritise fuzzing multiple targets by dynamically adjusting its focus based on the observed progress and the relative difficulty of reaching each target.

WAFLGo [25] is a directed greybox fuzzer designed to efficiently test commit change sites and their affected code, introducing a novel critical code guided input generation strategy and a lightweight multi-target distance metric. These innovations help ensure comprehensive testing by identifying and focusing on critical code paths and data dependencies. Despite its effectiveness, WAFLGo faces significant limitations that make it less suitable for integration into CI/CD pipelines. The substantial computational overhead due to intricate distance computations and critical code identification processes demands considerable time and resources, which are often not feasible within the tight time constraints of CI/CD environments. Although WAFLGo introduces a novel method for handling multiple targets, the complexity of maintaining individual directness for each target adds to the computational burden, resulting in slower performance and challenging the quick feedback cycles required in CI/CD workflows.

As discussed in the introduction, the evaluation of prior directed fuzzing techniques does not take into account the pre-computation stages, particularly the distance calculation, as part of the fuzzing budget [3, 6, 11, 15, 27]. This is an unrealistic assumption, and an important bottleneck in terms of deploying these techniques within CI/CD workflows.

TargetFuzz [5] proposes a mechanism to enhance directed greybox fuzzing by using a target-oriented seed corpus (DART corpus), which contains 'close' seeds to the targets. This approach significantly improves bug-finding capability and efficiency, by providing a complementary enhancement to directed fuzzing rather than altering the main algorithm.

While AFLChurn [28] focuses on fuzzing recently changed code to catch regressions, it operates under the principle of targeting new code changes with higher priority to ensure that recent modifications do not introduce new bugs. This approach is particularly effective for regression testing but does not specifically address the scalability challenges of directed greybox fuzzing within CI/CD pipelines.

In the same spirit, Klooster et al. [12] propose to use fuzz targets impacted by the recent code changes in order to reduce fuzzing overhead. However, this still uses undirected fuzzing and hence does not try to make the CI/CD fuzzing more directed.

In addition to greybox fuzzing, symbolic execution [4] has also shown promise in the context of patch testing. Works such as KATCH [17] based on dynamic symbolic execution target the problem of patch reachability, while Noller et al. [18] and SHADOW [19] build on top of dynamic symbolic execution to comprehensively *exercise* software patches after reaching their code.

7 Conclusion

We have presented PaZZER, a tool designed to optimise the scalability of directed greybox fuzzing within CI/CD pipelines by building on the foundation established by AFLGo. By addressing the key bottleneck of distance computation to lines of code affected by patches, we proposed two inter-related optimizations: eliminating one type of distance computation and implementing an incremental algorithm for distance calculation.

Through preliminary experiments, we have demonstrated that dropping the CFG based distance computation does not significantly affect the *directedness* and hence the effectiveness of the fuzzing process. Additionally, our incremental algorithm shows promising results in reducing the time required for distance computations, making it feasible for short per-commit fuzzing runs.

We have outlined a comprehensive plan for further experimental evaluation to rigorously assess the effectiveness of PaZZER on a diverse set of applications. This evaluation will measure the improvements in fuzzing efficiency and effectiveness in the context of CI/CD pipelines, focusing on metrics such number of bugs found, code coverage, and overall performance compared to the original AFLGo.

By addressing the scalability challenges inherent in directed greybox fuzzing, PaZZER aims to make per-commit fuzzing campaigns more practical and effective, ultimately contributing to more reliable and secure software systems.

8 Data-Availability Statement

Our artifact is currently available at [22].

9 Acknowledgements

This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141).

References

[1] [n. d.]. GIFLIB: A Library for Reading and Writing GIF Images. https://giflib.sourceforge.net/. Accessed: 2024-06-11.
[2] [n. d.]. Libming: SWF Output Library. https://github.com/libming/libming. Accessed: 2024-06-11.
[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *CCS'17* (Dallas, TX, USA). https://doi.org/10.1145/3133956.3134020
[4] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *CACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795
[5] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. 2022. TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 561–573. https://doi.org/10.1145/3488932.3501276
[6] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*. 2440–2451.
[7] Free Software Foundation, Inc. [n. d.]. GNU Binutils. https://www.gnu.org/software/binutils/.
[8] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
[9] M Jansen and Michael Buro. 2007. HPA* enhancements. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 3. 84–87.
[10] JasPer [n. d.]. JasPer. https://www.ece.uvic.ca/~frodo/jasper/.
[11] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948.
[12] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. 25–32.
[13] Sven Koenig and Maxim Likhachev. 2002. D*lite. In *Eighteenth National Conference on Artificial Intelligence* (Edmonton, Alberta, Canada). American Association for Artificial Intelligence, USA, 476–483.
[14] Sven Koenig, Maxim Likhachev, and David Furcy. 2004. Lifelong Planning A*. *Artificial Intelligence* 155, 1 (2004), 93–146.
[15] Hongliang Liang, Xinglin Yu, Xianglin Cheng, Jie Liu, and Jin Li. 2023. Multiple targets directed greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2023).
[16] Maxim Likhachev, David I Ferguson, Geoffrey J Gordon, Anthony Stentz, and Sebastian Thrun. 2005. Anytime dynamic A*: An anytime, replanning algorithm.. In *ICAPS*, Vol. 5. 262–271.
[17] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *ESEC/FSE'13* (Saint Petersburg, Russia). https://doi.org/10.1145/2491411.2491438
[18] Yannic Noller, Hoang Lam Nguyen, Minxing Tang, and Timo Kehrer. 2018. Shadow Symbolic Execution with Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 42, 4 (2018), 1–5.
[19] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences Between Software Versions. In *ICSE'16* (Austin, TX, USA).
[20] Ganesan Ramalingam and Thomas Reps. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21, 2 (1996), 267–305.
[21] Kostya Serebryany. 2017. OSS-Fuzz – Google's continuous fuzzing service for open source software. In *USENIX Security'16* (Vancouver, BC, Canada). Invited talk.
[22] Arindam Sharma, Cristian Cadar, and Jonathan Metzman. 2024. Artifact of Effective Fuzzing within CI/CD Pipelines (Registered Report). https://doi.org/10.6084/m9.figshare.26075962.v3. Figshare.
[23] Continuous Integration through OSS-Fuzz. 2024. *CIFuzz*.
[24] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 789–800.
[25] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Xiao Xiao, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical Code Guided Directed Greybox Fuzzing for Commits. In *USENIX Security'24* (Philadelphia, PA, USA).
[26] Michal Zalewski. [n. d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[27] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 1343–1360.
[28] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2169–2182.

Received 2024-06-21; accepted 2024-07-22