POGOFUZZ: Profile-Guided Optimization for Fuzzing (Registered Report)

Tobias Holl
Ruhr University Bochum
tobias.holl@rub.de

Leon Weiß
Ruhr University Bochum
leon.weiss@rub.de

Kevin Borgolte
Ruhr University Bochum
kevin.borgolte@rub.de

***Abstract*—Fuzzing is one of the most successful techniques to test software and discover vulnerabilities. Due to its effectiveness and ease of scaling, it is often done in parallel on hundreds to thousands of CPU cores and improving fuzzers' efficiency, efficacy, and performance has become a major research area. Typically focused on enhancing fuzzing itself, such as through better input generation or optimizing instrumentation to execute the program more frequently, the goal is to find more bugs or flaws in less time. On the other hand, optimizing the performance of the target program, which the fuzzer executes billions of times, specifically for fuzzing has received little attention.**

We introduce POGOFUZZ, a novel approach to improving fuzzing performance that is *fuzzer-agnostic* and *target-agnostic*. We leverage the insight that the inputs used for future mutations are known, to then use compiler-based profile-guided optimization (PGO) to optimize the target program specifically for these future inputs. By regularly creating new profiles based on the next inputs, recompiling the target program with new optimizations, and *in-situ* replacing the target in the fuzzing process with its newly optimized version, POGOFUZZ improves fuzzing performance of the state-of-the-art fuzzer AFL++.

We provide preliminary results for POGOFUZZ in different realistic experimental setups, comparing it to AFL++ on four software projects from the FuzzBench suite for 1–6 physical CPU cores per fuzzer, to demonstrate POGOFUZZ's advantages. Our preliminary results show that our approach has the potential to improve fuzzing throughput, despite incurring additional optimization and recompilation costs. POGOFUZZ, as a *fuzzer-target-agnostic* approach, is a significant departure from traditional improvements in fuzzing, which are fuzzer-specific and/or target-specific, providing the opportunity for new, general performance improvements for large-scale, extended fuzzing.

To encourage adoption and reproducibility of our research, we will make POGOFUZZ publicly available as open source before or with the publication of the extended paper.

I. INTRODUCTION

Fuzzing has become a major, if not the main, approach for vulnerability discovery, after showing a remarkable efficacy in uncovering bugs. Most modern fuzzers are coverage-guided, using greybox analyses to navigate the target program's code coverage space "intelligently," often based on AFL [15] or AFL++ [9]. They iteratively refine their input generation to increase code coverage. Recent work further improves fuzzing, for example, by combining it with symbolic execution [28], taint analysis [23], or machine learning [26].

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23029
www.ndss-symposium.org

The real-world impact of fuzzing has been profound and it has been widely adopted by practitioners. Large-scale fuzzing has become common, often utilizing vast computational resources that would sit idle otherwise. A prime example is OSS-Fuzz, an initiative by Google for fuzzing open source software, which uses hundreds of thousands of processor cores [21]. It supports the simultaneous, continuous fuzzing of over 1,300 open source projects, allocating an average of over 80 cores per project to fuzzing. The true scale of fuzzing in industry almost certainly extends far beyond the known open source initiatives like OSS-Fuzz, with major companies investing heavily in private fuzzing infrastructure [16, 18]. Already in 2019, Google's infrastructure for fuzzing only Google products used more than 25,000 cores [2]. Microsoft is also fuzzing its products extensively [10].

Given the large scale and resource-intensive nature of fuzzing, its performance and efficacy has become a focus of research. Improving performance by increasing the number of target executions per second directly implies a higher chance of discovering bugs, because more inputs are tested in the same amount of time. Prior work has largely focused on improving or introducing fuzzer components, like snapshot-based [1, 25, 30] or persistent-mode [9] fuzzing. Meanwhile, optimizing the performance of the target programs specifically for fuzzing has only received little attention [20]. However, optimizing the target presents a unique opportunity to improve efficiency, as the programs are executed for long periods of time and often tens of thousands of times per second on every single processor core. Nagy et al. [20] proposed the removal of unnecessary instrumentation from the target, and have shown the potential benefits of optimizing the target instead of the fuzzer, but it remains an open research question if and how we can comprehensively optimize a target specifically for fuzzing.

In this paper, we introduce POGOFUZZ, a novel approach to improve fuzzing performance that is both *fuzzer-agnostic* and *target-agnostic*. It leverages the insight that, in fuzzing, the future inputs are mutations of known inputs, and they likely exercise the same code paths. By creating corresponding program profiles, POGOFUZZ can use profile-guided compiler optimizations (PGO) to optimize the target specifically for the future inputs, and *improve fuzzing performance independent of the fuzzer and agnostic to the target.*

Since the input seeds change over time, for example, after discovering new code coverage or marking other inputs as more interesting for the future, POGOFUZZ regularly needs to

create new profiles, recompile the target with new optimizations, and *in situ* replace it with its newly optimized version without requiring costly fuzzer restarts.

Fundamentally, POGOFUZZ is a significant departure from traditional fuzzer optimizations, and offers the potential for substantial performance gains in large-scale, extended fuzzing. We make the following contributions:

- We introduce POGOFUZZ, a *fuzzer-agnostic* and *target-agnostic* approach to improve fuzzing performance, which leverages the insight that the seeds to generate future inputs are known, allowing us to optimize the target via profile-guided optimizations (PGO).
- We evaluate POGOFUZZ in various experimental settings that reflect smaller, real-world fuzzing efforts, ranging 1–6 physical CPU cores per fuzzer, and compare it to the state-of-the-art fuzzer AFL++ on four real-world targets.
- We provide preliminary evidence that POGOFUZZ can indeed improve fuzzing performance over AFL++ by up to 3.7% median (up to 29.7% max), and that, even at frequent recompilation intervals, POGOFUZZ's improved performance amortizes its recompilation costs.

II. BACKGROUND

A. Profile-Guided Optimization

Modern, optimizing compilers rely on a number of analyses and heuristics to make decisions on which transformations to perform. Estimates of how often parts of the code will be executed influence function inlining, code motion within functions, hot-cold-splitting, loop unrolling, and other optimizations. In particular, compilers will aggressively optimize code for speed that they expect to be executed often (*hot path*), while trading speed for size in rarely executed, *cold* code.

While these estimates can sometimes be easy to obtain statically (e.g., for fixed iteration count loops), compilers are generally unable to reason about how exactly the program will be used in practice. For code paths that heavily depend on the input data, such as file format parsers, optimization decisions may be suboptimal or even harmful to performance if the compiler's estimate is too far removed from real-world inputs.

Profile-guided optimization (PGO) uses dynamically obtained feedback (*profile*) to inform the compiler how often each part of the code is executed, typically collected from real-world workloads. In turn, the compiler can make better decisions during optimization, which should lead to better performance. Typically, the compiler inserts profiling instrumentation during compilation, which records how often each instrumentation point is executed [7]. This is similar to the coverage instrumentation used by fuzzers, but tracks exact execution counts instead. To enable further optimizations, modern compilers can also record additional information. For example, LLVM can record *value profiles* that contain information on the sizes of certain memory operations, like `memcpy`, which may benefit from vectorization if they are large enough.

By collecting this data under realistic workloads, in real deployments or with synthetic benchmarks, the profile will be representative of the program's 'typical" use. It is then recompiled, with the profile replacing the compiler's original estimates of execution frequencies, branch probabilities, etc.

Figure 1 shows a common PGO workflow: The compiler instruments the program and compiles it using an initial estimate of the hot path, which affects the optimizations made during compilation (in Figure 1, we illustrate the impact on basic block ordering). After profiling, the compiler knows the actual, real execution frequencies and it can make (much) more accurate optimization decisions.

Confusingly, *profile-guided optimization* is sometimes used interchangeably with *feedback-directed optimization* (FDO). In the context of LLVM however, PGO is typically used to describe instrumentation-based methods, while FDO refers to approaches in which execution frequencies are obtained by sampling instead (e.g., AutoFDO [4]). Nowadays, PGO and FDO are widely used to optimize large, real-world software, like the Android operating system [4, 29] or the Chromium browser [6]. PGO, FDO, and similar adaptive reoptimization approaches are also common in language interpreters. For example, JavaScript engines, such as V8, make runtime decisions on whether to perform just-in-time compilation and how aggressively to optimize based on runtime execution counts. In dynamic binary translation, optimizing frequently executed code can also lead to significant performance gains [13]. Further, individual optimizations can also be feedback-driven (e.g., using observed type frequencies to optimize dynamically typed code). Of course, generally, the performance gains for compiled code will be less than when deciding whether to compile at all, but can still be considerable.

III. POGOFUZZ

POGOFUZZ is a new *fuzzer-agnostic* and *target-agnostic* approach to improve fuzzing performance by optimizing the target program for its future concrete inputs. It extends an existing fuzzer, like AFL++ [9], with a recompilation stage that derives PGO data from the current inputs and uses it to reoptimize the target, shown in Figure 2. Since the nature of the inputs change over time, we need to regularly reoptimize. In the following, we describe the design of POGOFUZZ.

A. Input Selection

Generally, a fuzzer decides dynamically *which queue entries* to mutate, and *how many new test cases* to generate from an input. This decision is typically based on a *power schedule* [3]. We use this energy metric to determine which entries will be mutated and passed to the target. We then optimize the target program so that it executes these test cases especially quickly, with the idea that this should speed up the fuzzing process by increasing the number of target executions per second.

We scale proportionally with the number of new test cases that will be generated from each queue entry by the fuzzer. Since the fuzzer will not mutate test cases with a low score to create a significant number of new inputs, we skip low-energy queue entries for our reoptimization using a simple threshold. In turn, we can avoid processing the entire input

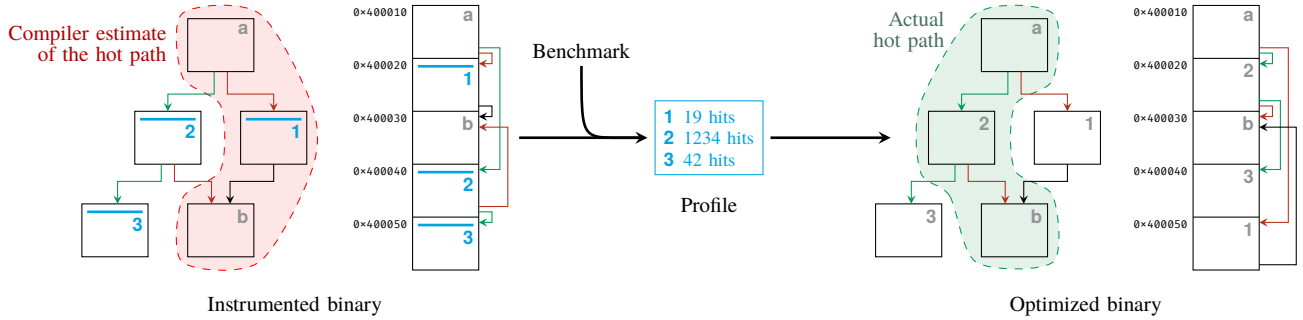


Figure 1: Overview of the profile-guided optimization (PGO) workflow. Here, we illustrate the effect on basic block ordering, but PGO also influences many other optimizations.

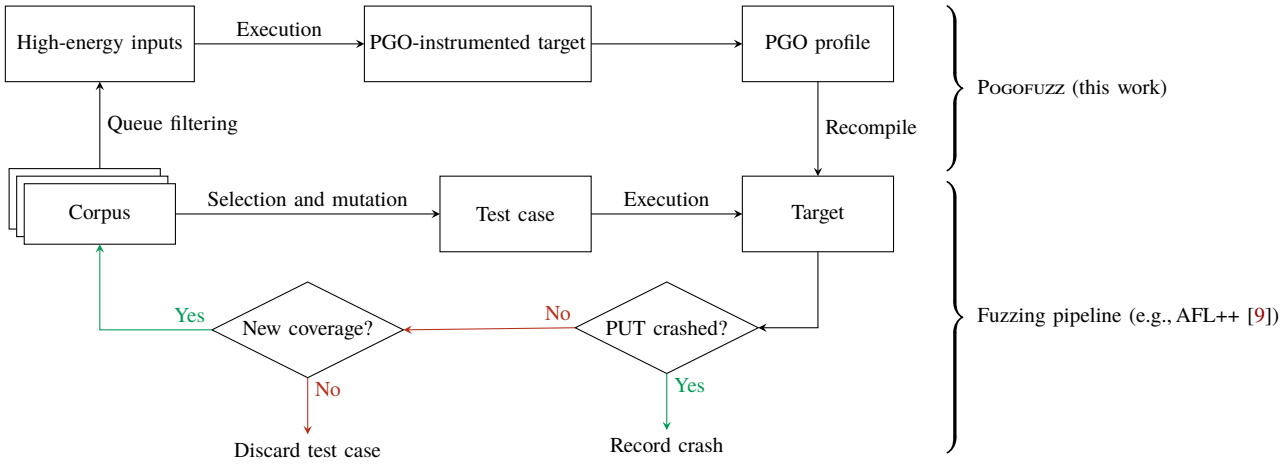


Figure 2: High-level architecture of PogoFuzz.

queue, which may take prohibitively long. Instead, we focus on the subset of high-energy inputs that will yield the majority of new mutations. While the energy of a particular test case will decay over time, the nature of the test cases generated during fuzzing generally changes slowly. Moreover, only when there is a significant change in coverage (and therefore at least one new, interesting, high-energy test case), or when the fuzzer has had sufficient time to explore high-energy inputs does it become useful to reoptimize the target. Thus, the time between reoptimizations provides an opportunity to exploit the better performance and amortize the recompilation cost.

B. Profile Generation

After we select the high-energy inputs that will be used to test the target, we derive PGO profiles for these inputs using a separately instrumented target binary, and we then use the merged profile to reoptimize the target.

In our prototype of PogoFuzz, we use a variant of LLVM's IR-level PGO instrumentation, modified to scale with the test case's energy level, to derive profiles for the high-energy inputs. Because our profiles are compatible with LLVM, we can reuse existing compiler infrastructure to optimize during recompilation. This allows PogoFuzz to be readily integrated with any fuzzer and target that can use LLVM for compilation, without the need for target-specific modifications. With minimal engineering effort, PogoFuzz can be also be ported to other compilers with PGO support.

We want to add our PGO instrumentation late in the compilation process, so that our PGO binary is as close to the fuzzer's target binary as possible. To ensure this, we use LLVM's IR-based PGO instrumentation instead of its front-end instrumentation (which more accurately reflects the source-code level). To ensure that our generated profiles match the fuzzer's instrumented binary as closely as possible, we keep the build pipeline constant between recompilations by using the same configuration used to compile the target for fuzzing. Fortunately, we can avoid faulty estimates: LLVM's PGO implementation is inherently fault-tolerant, because it overrides compiler estimates with PGO data only for functions that did not structurally change (according to a hash of the control flow graph) between instrumentation and reoptimization.[1]

We further adapt concepts from fuzzer design to make PogoFuzz's profile generation process more efficient. Instead of manually invoking the profiling binary for every high-

[1]The compiler produces warnings about functions with such structural changes. We did not encounter any such warnings during our evaluation.

energy test case, we use a fork server (derived from AFL++'s fork server instrumentation) to efficiently evaluate each input. After a test case finishes, we accumulate the execution counts for every instrumented edge, which implies that we cannot naively use post-processing to scale the counts by each input's individual weight. Instead, to avoid having to repeatedly evaluate an input for proper weighting, we use a custom compiler pass to modify the profiling instrumentation:

1) We locate every program point at which LLVM inserted profiling instrumentation. We do this by inspecting the LLVM *intermediate representation* (IR) and searching for calls to the `llvm.instrprof.*` intrinsics. This must be done immediately after they are inserted, because the default LLVM optimization pipelines will lower these intrinsics to normal IR code in the next few passes.
2) We replace all `llvm.instrprof.increment` calls with `llvm.instrprof.increment_step` calls. The latter intrinsic is a stepped variant of the former that lets us specify by how much the counter should be incremented.
3) We insert new code to load the weight of the current test case before calls to `llvm.instrprof.increment_step`. We then multiply the old increment by our weight (i.e., the new increment is the weight for old increments by 1 and the old value multiplied by the weight otherwise).

C. Recompilation and Replacement

Next, we use our profile to recompile and reoptimize the target, and then replace the target *in situ* in the fuzzing process.

To enable recompilation, PogoFuzz adds a custom recompilation stage to the fuzzer that triggers on a configurable timer. For our preliminary evaluation, we recompile every hour. We decide to recompile every hour because it balances the length of a typical fuzzing campaign and how long the top queue entries are being fuzzed with the typical build time of a target (approximately a few minutes). Other recompilation intervals, like 2–6 hours, but also recompilation intervals that are dynamic or tailored to a target, might allow for larger performance gains and should be investigated.

Over time, high energy inputs become less important because they were executed sufficiently often. However, we also want to avoid recompiling too frequently, as it incurs additional cost, which we might not be able to recoup. For example, if recompilation on a single core takes 2 minutes and we recompile every hour, then we fuzz only for 58 minutes. If our optimized target does not reach (on average) 3.45% more executions per second, we cannot amortize our recompilation cost. Naturally, with more CPU cores, our gains scale and it becomes easier to recoup this cost, which makes PogoFuzz particularly well-suited for large-scale, real-world fuzzing.

When PogoFuzz triggers recompilation, we recompile the fuzzer target from scratch using the same build process and configuration as before, but we now also use our profile for optimization. We utilize one fuzzing process on one of the CPU cores for recompilation and stop fuzzing on it, while fuzzing processes on all other cores continue fuzzing the current target binary until we compiled the reoptimized version. Once we compiled the new version, we replace the target binary *in situ*, and the instance of the fuzzer responsible for recompilation signals all other fuzzing processes to continue fuzzing with the recompiled target. This means that these processes will start fuzzing the reoptimized target binary with minimal overhead.

IV. Preliminary Evaluation

We focus our preliminary evaluation of PogoFuzz on executions per second and compare against AFL++ v4.32c, specifically `afl-clang-lto`, which PogoFuzz is based on. For our preliminary investigation, we study a subset of four targets from FuzzBench [17]. We selected these four targets based on two criteria likely affecting PogoFuzz the most, namely their executions per second and their input queue size:

- libxml2 (`libxml2_xml` in FuzzBench) is a relatively slow target (in observed executions per second), and produces a large queue that may need to be evaluated for profiling.
- libxslt (`libxslt_xpath` in FuzzBench) also produces a large queue, but executes significantly faster than libxml2.
- systemd (`systemd_fuzz-link-parser`) is another slow target, but only produces a small number of queue entries.
- zlib (`zlib_zlib_uncompress_fuzzer`) has the highest execution speed of the four, but also only produces a small number of queue entries.

We compare PogoFuzz and AFL++ in multiple configurations, ranging 1–6 physical CPU cores per fuzzer, to determine when the speedup achieved by reoptimization outweighs the cost of the recompilation, that is, to investigate the effect of scaling. For statistical rigor, we repeat each experiment 5 times and report median, standard deviation, and maximum.

We performed our preliminary experiments on two Dell R7625 servers with two AMD EPYC 9754 CPUs each (with 128 physical cores per CPU, i.e., 256 physical cores per server and 512 physical cores total) and 2048 GiB of DDR5 system memory each. Both servers ran Debian 12. We assign targets to servers, that is, we evaluate both fuzzers for two targets on one server, and both fuzzers for the other two targets on the other server. To minimize the impact of hardware behavior and ensure that there is no unintended interference between fuzzing trials, we disabled hyperthreading and any settings that attempt to dynamically scale hardware performance at runtime. This includes dynamic frequency scaling for the CPU ("Turbo Boost") and for the interconnects (e.g. AMD's "Algorithm Performance Boost"). We used only 240 of 256 physical cores per server for fuzzing, with the remaining 16 cores remaining available for the OS and orchestration. We pinned each fuzzing processes to a unique physical core via AFL++'s builtin CPU affinity settings, and we restrict the individual trials to the cores allocated to their fuzzing processes using `cpuset` cgroups and `numactl`. Further, to eliminate the impact of disk contention, we stored the fuzzer's input queue and experiment results entirely in memory via `tmpfs`, with the fuzzers and target binaries being stored on NVMe SSDs.

A. Results

Figure 3 shows the number executions per second of POGOFUZZ and baseline AFL++ over 24 hours and Table I reports the total executions after 24 hours.

As expected, reoptimization imposes a greater performance penalty when only few cores are used for fuzzing. After all, one core is occupied with compilation and cannot be used for fuzzing during that time. This behavior is particularly visible in the results for libxslt (Figure 3b) and systemd (Figure 3c), where single core execution speed almost drops to zero for short periods. For example, recompiling systemd during a single-core trial leads to a loss of over 10% of fuzzing throughput (median 8.5×10^6 executions instead of 9.5×10^6 executions on baseline AFL++). This corresponds closely to the portion of total time taken up by the recompilation process.

As the number of cores increases, the reoptimization cost becomes less significant and amortizes quickly. Some targets are more sensitively to reoptimization and show the performance gains of POGOFUZZ with fewer cores than others. Starting from around three to four cores, POGOFUZZ consistently outperforms AFL++ on libxml2 and libxslt by up to 3.7% in the median. This trend continues if we perform an additional experiment using 8 cores (see Figure 4). On systemd, which has very few executions per second in general (≈100 executions per second per core), the performance of AFL++ and POGOFUZZ is within the margin of error. With zlib, median execution speeds are generally below or equal to the baseline until we perform an additional experiment with 8 cores, where we see clear performance gains over AFL++ (see Table I and Figure 5). This naturally raises the question of how well POGOFUZZ's gains scale with more physical cores, which we will investigate (see Section V-B).

Table I: **Total executions per target per fuzzer after 24 hours, in million.** The largest median number of executions per target and core configuration is bolded. Particularly notable are the results for libxslt, for which the median number of executions of POGOFUZZ with 8 cores exceeds the performance of baseline AFL++ by 5σ, and zlib with 8 cores, for which POGOFUZZ achieves 165 million more executions than the baseline in 24 hours (20.6 million executions per core).

		Baseline AFL++			**POGOFUZZ**		
Target	**Cores**	Median	σ	Max	Median	σ	Max
libxml2	1	100.7	29.9	120.0	**104.0**	28.2	137.2
	2	192.8	59.6	318.8	**214.4**	78.6	334.6
	3	304.5	88.3	439.7	**382.9**	62.9	431.7
	4	403.1	80.4	437.3	**413.0**	95.1	567.0
	6	762.2	213.0	975.7	**793.2**	124.7	1001.0
libxslt	1	360.6	10.2	368.4	**362.3**	9.5	372.5
	2	**731.6**	17.0	759.3	707.4	45.6	764.5
	3	1121.9	13.9	1137.2	**1149.6**	19.1	1178.7
	4	1476.8	19.9	1498.6	**1520.1**	23.9	1547.7
	6	2260.7	33.3	2278.2	**2290.0**	37.3	2319.1
	8	2985.1	37.3	3020.9	**3096.8**	14.9	3100.6
systemd	1	**9.5**	0.1	9.7	8.5	0.1	8.6
	2	**19.9**	0.6	21.0	19.1	0.6	20.0
	3	29.8	0.8	30.7	**29.9**	0.7	30.6
	4	**40.8**	2.0	43.4	38.8	0.7	40.2
	6	**61.4**	1.4	62.2	60.9	2.4	63.8
zlib	1	631.8	129.3	746.9	**670.7**	21.8	707.0
	2	**1540.9**	25.9	1592.4	1495.9	20.1	1528.2
	3	2312.8	46.0	2342.4	**2330.2**	109.3	2375.8
	4	**3160.1**	16.9	3192.2	3137.8	8.6	3145.2
	6	**4758.3**	67.7	4898.8	4694.2	51.8	4771.6
	8	6337.6	62.3	6377.1	**6407.2**	135.7	6542.5

V. DISCUSSION

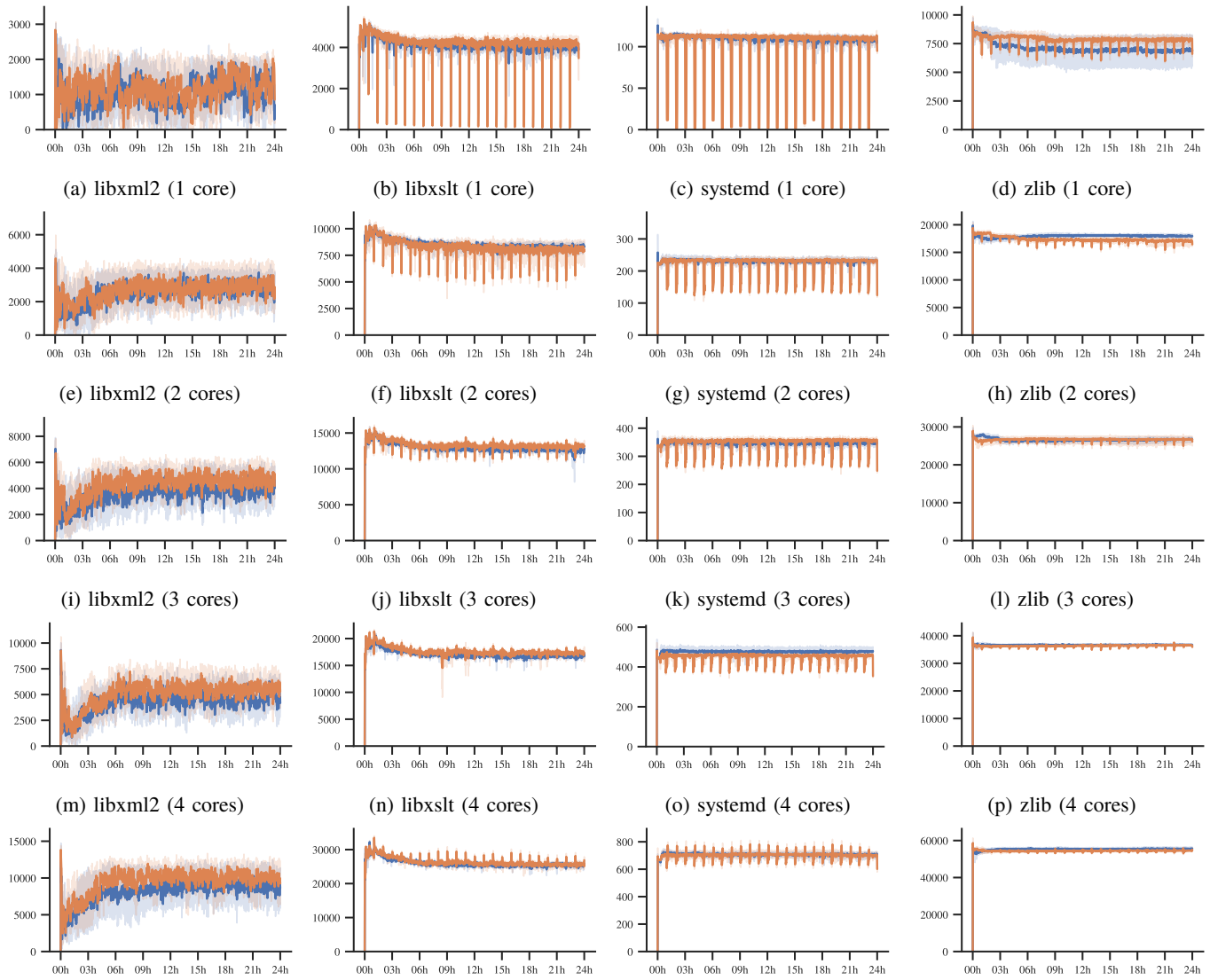
A. Possible Limitations and Threats to Validity

In this work, we did not examine whether recompilation may introduce changes in the mapping between code locations and the fuzzer's coverage map. That is, whether the locations of edges in the coverage map before and after recompilation are the same. The primary reasons for such a change to occur are changes to the program structure caused by different optimization decisions, and changes introduced by the instrumentation logic. Thankfully, changes to program structure that are relevant to the fuzzing instrumentation are very rare and would mainly be caused by inlining decisions made early in the compilation pipeline. Most other optimization decisions that PGO-derived data informs are made after the fuzzing instrumentation is inserted, especially so if we use IR-based PGO instrumentation. This includes machine-specific instruction selection and code positioning (e.g. basic block ordering). Given that POGOFUZZ leverages LLVM's IR-based PGO instrumentation and its fault-tolerance, we do not expect such issues to occur for us and we did not observe any during our preliminary analysis. However, such issues might occur with other compilers' PGO approaches.

Another potential source for changes to the coverage mapping that could cause an issue is the fuzzer's instrumentation pass itself. This would require the instrumentation to take the optimization metadata provided by PGO into account though. AFL++ does not do this in any of its plugin-based instrumentations, including `afl-clang-lto`. In fact, it is very difficult to use such sophisticated approaches to selecting instrumentation sites in a fuzzer. Thus, we do not expect there to be any changes to the fuzzer's coverage mapping.

Generally, other instrumentation methods for PGO might produce different results. We opted for LLVM's IR-based PGO instrumentation because it is widely used and we expect front-end instrumentation to perform worse, but it is possible that other options or compilers may produce profiles of higher fidelty, leading to even greater improvements. Of particular interest are sample-based profiling methods like AutoFDO [4] that could be used throughout the fuzzing campaign (though we do not need to observe real-world inputs in the first place, as we can know the nature of the inputs the fuzzer will create), and LLVM's context-sensitive PGO instrumentation (CSIR), though its context tracking does not lend itself well to weighting during reinstrumentation (see Section III-B).

To balance recompilation cost with expected changes to the input queue over time and the performance gain of PGO, we recompile the target once per hour. However, recompiling



(a) libxml2 (1 core) (b) libxslt (1 core) (c) systemd (1 core) (d) zlib (1 core)

(e) libxml2 (2 cores) (f) libxslt (2 cores) (g) systemd (2 cores) (h) zlib (2 cores)

(i) libxml2 (3 cores) (j) libxslt (3 cores) (k) systemd (3 cores) (l) zlib (3 cores)

(m) libxml2 (4 cores) (n) libxslt (4 cores) (o) systemd (4 cores) (p) zlib (4 cores)

(q) libxml2 (6 cores) (r) libxslt (6 cores) (s) systemd (6 cores) (t) zlib (6 cores)

Figure 3: Executions per second for libxml2, libxslt, systemd, and zlib baseline AFL++ (●) and PoGoFuzz (●)

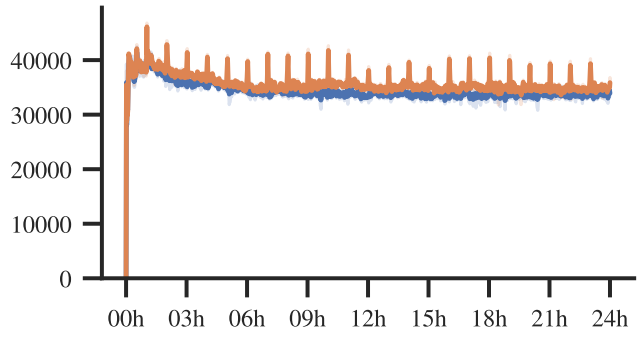


Figure 4: Executions per second over 24 hours for libxslt (8 cores) for baseline AFL++ (●) and PoGoFuzz (●)

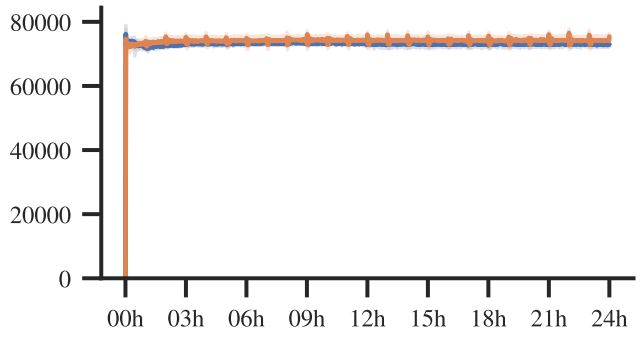


Figure 5: Executions per second over 24 hours for zlib (8 cores) for baseline AFL++ (●) and PoGoFuzz (●)

less frequently, target-dependent, or dynamically based on how much the set of high-energy input test cases differs from the input set previously used for reoptimization may yield even better improvements. We will investigate other recompilation intervals in our follow-up experiments (see Section V-B).

Finally, as is often the case with compiler optimizations, target-specific variations can lead to differences in the effectiveness of profile-guided optimization. To provide an initial assessment whether PogoFuzz might be useful for real-world fuzzing, we evaluated it on four programs that exhibit distinctly different behavior in execution speed and input queue size, the two main factors influencing PogoFuzz's efficacy. Nevertheless, to ensure the generalizability of our findings, we will evaluate PogoFuzz on a broader range of programs (see Section V-B). Similarly, PogoFuzz's scalability and benefits for larger-scale fuzzing should be studied, on the scale of dozens to hundreds of cores per target (e.g., for OSS-Fuzz).

B. Proposed Experiments

For our preliminary evaluation, we used a recompile interval of 1 hour. This fixed interval is likely not optimal for all targets, but it does allow us to clearly show the potential of PogoFuzz and that the additional cost of recompiling the target can be recouped even at frequent intervals and when performing likely unnecessary recompilations. At less frequent recompilation intervals, for example, every 2 hours, PogoFuzz's recompilation overhead would reduce, but the target's executions per second may degrade if the input set changes substantially, that is, if the inputs do not exercise the optimized target's hot code paths anymore. Dynamically determining when recompilation should occur, per target, will almost certainly lower PogoFuzz's recompilation overhead, while also retaining most or all of the performance gains that PogoFuzz contributes, which is a question that we will investigate. In particular, we will:

- We will evaluate different recompile intervals of at least 2, 4, and 8 hours for fuzzing campaigns of 48 hours.
- We will explore how one can use heuristics to skip regular recompilation intervals if the queue has not changed substantially from the last recompilation, or how one could dynamically determine the best time for recompilation.

So far, we evaluated PogoFuzz in experimental settings that reflect how fuzzers are run for smaller scale fuzzing campaigns. Already, we observed performance gains over baseline AFL++ at 4–8 cores per fuzzer. However, in practice, fuzzing campaigns can span tens to hundreds to thousands of cores per fuzzer. We expect the performance gains of PogoFuzz to further increase, as fuzzing is perfectly parallel and the gains should scale (almost) linearly. Moreover, while we have shown that PogoFuzz increases performance for an example of each of the four classes of fuzzing targets (cf. Section IV), that is, slow or fast targets with small or large queues, the performance characteristics of other targets of these classes or other classes may differ. Therefore, we will extend our experiments and confirm our preliminary hypotheses:

- We will allocate (many) more computational resources to each fuzzer, up to (at least) 24 CPU cores per fuzzer trial.
- We will evaluate and compare PogoFuzz on all targets of the FuzzBench benchmarking suite.

Further, for our preliminary analysis, we repeated all our configurations of fuzzer, target, number of cores for 5 times and report median, standard deviation, and maximum values. We will extend our experiments and implement best practices for fuzzing evaluations [24]:

- We will extend our fuzzing duration to (at least) 48 hours.
- We will run (at least) 10 trials per fuzzer configuration.
- We will perform robust statistical significance testing and report the effect size.

Finally, to encourage adoption and reproducibility, we will make PogoFuzz and related artifacts publicly available as open source at `softsec.link/fz26.pogofuzz`.

C. Opportunities for Future Work

PogoFuzz improved fuzzing performance over the highly optimized state-of-the-art fuzzer AFL++, but our approach is fuzzer-agnostic: It does not modify or require any fuzzer components. The performance improvements over other fuzzers, like Honggfuzz [14], may differ. However, PogoFuzz does not improve the performance of the fuzzer itself, but only that of the target, which strongly suggests that the performance gains would readily transfer to other fuzzers.

Our implementation of PogoFuzz is a proof of concept and not highly optimized. Various improvements that have been made to fuzzing and compiler infrastructure could be adopted to our approach to improve the performance of PogoFuzz itself, rather than that of the target. For example, it may be possible to tune the performance of the recompilation step or to cache parts of it before the PGO profile is loaded, similar to AFL++'s forkserver, but adapted to the PGO process. We refrained from these optimizations as the compilation process takes a relatively small amount of time and we observed that PogoFuzz's overhead is amortized for few CPU cores already. However, future work investigating the possible gains on targets that have long compilation times, such as web browsers or operating systems, may be worthwhile.

So far, we leverage compiler-based PGO to optimize the target program, but other optimization techniques such as post-link optimizations (PLO) that perform (static) binary rewriting, like BOLT [22] or Propeller [27] could yield additional improvements. Both of them have shown promise in providing additional performance improvements over PGO for large-scale data-center binary programs, and they might allow for additional performance improvements, but also at an additional cost. While the resources to properly evaluate PLO's potential are beyond what is available to academic researchers, future work should investigate the necessary scale at which the PLO cost, in addition to PGO, can be amortized and when PLO can improve fuzzing performance (e.g., at the scale of OSS-Fuzz).

VI. Related Work

Prior work related to PogoFuzz falls in two main research areas: approaches to optimize fuzzer performance (Section VI-A), and improvements and applications of profile-guided optimization (Section VI-B).

A. Optimizing Fuzzing Performance

Fuzzing and improving its efficacy has received substantial attention from the security and software engineering communities in recent years. A core focus has been on reaching more code, that is, achieving higher code coverage, in the same time.

Closely related to PogoFuzz are approaches aiming to reduce overhead. Zhang et al. [32] studied system-level optimizations to accelerate fuzzing. They found that common fork server and OS operations can slow down fuzzing significantly, and propose replacing slow components with optimized ones that avoid these costly interactions when possible. PogoFuzz instead optimizes the target and does not require modifications to the operating system or fuzzer, making it complementary.

Snapshot-based fuzzing eliminates the need to reinitialize the target each time by recording the state at a program point and later restoring it for each new input, which reduces overhead and increasing execution speed. For example, Nyx [25] is a snapshot-based hypervisor fuzzer. PogoFuzz is largely complementary to snapshot-based fuzzing, with the caveat that it may be necessary to reset the snapshot after recompilation. Considering that snapshot-based fuzzers typically achieve higher executions per second speeds than traditional fuzzers and we can prioritize reoptimizing code after the snapshot, PogoFuzz's benefits might be even larger.

Prior work has also aimed to identify and remove unnecessary instrumentation to improve performance. Zhang et al. [31] introduce ASan--, a "debloated" Address Sanitizer (ASan) [11]. ASan is commonly used to instrument targets to detect memory errors, but it incurs substantial runtime overhead. ASan-- incurs up to 70% less overhead than ASan, while still detecting most memory errors accurately. Similarly, Nagy et al. introduce UnTracer [20], an approach that reduces overhead by removing coverage instrumentation that serves no purpose anymore after extended fuzzing and only retains instrumentation that can still lead to new code coverage. PogoFuzz is complementary to such approaches.

B. Profile-guided Optimizations

Profile-guided optimization (PGO), sometimes called profile-directed feedback (PDF) or feedback-directed optimization (FDO), is a compiler optimization that uses profiling information gathered during program executions to inform optimization decisions. It has proven particularly effective in improving the performance of large-scale, complex software. For example, PGO is used by Google Chrome [6], Mozilla Firefox [19], Linux [8], and various databases.

Most PGO research tries to collect profiling information at runtime in production at low overhead, focusing on sample-based profiling approaches [4, 5]. These approaches struggle to achieve the same performance gains as instrumentation-based PGO due to the lower data resolution, sparking attempts to increase fidelity. For example, He et al. propose CSSPGO, a context-sensitive sampling-based approach using pseudo-instrumentation to improve profile quality [12].

Other work focused on post-link optimizations (PLO), that is, optimizations after conventional compiler optimizations. BOLT by Panchenko et al. [22] performs PLO for large data-center binaries at Facebook through static binary rewriting. Shen et al. introduced Propeller [27], which is a relinking optimizer designed for warehouse-scale workloads. It addresses the scaling challenges for traditional post-link optimizers for large binaries and distributed build systems. Deployed at scale at Google, Propeller improves on compiler-based PGO by 1.1%. Approaches like BOLT and Propeller can readily be adapted to further optimize the fuzzing target, for example, by adding a PLO step to PogoFuzz after the recompilation. It is unknown though if the approaches' performance improvements also apply to fuzzing targets, which are typically much smaller than the large data-center binaries that these approaches were designed for (these binaries are often times 100MB and larger monolithic binaries and code layout is an important factor).

VII. Conclusion

We introduced PogoFuzz, a novel approach to enhancing fuzzing performance that is both fuzzing and target independent, by focusing on optimizing the performance of the target rather than the fuzzer. PogoFuzz uses profile-guided compiler optimizations (PGO) to specifically tailor the target to the inputs, leveraging the insight that the future inputs are likely exercising similar code paths and that the inputs are known a priori. This insight also eliminates the need to collect profiles at runtime with significant overhead, allowing us to reap PGO's benefits for fuzzing at almost no additional cost.

Through our preliminary evaluation of PogoFuzz on four real-world software projects from the FuzzBench benchmarking suite and with varying different computational resources (1–6 physical CPU cores per fuzzer), we have shown that PogoFuzz can surpass the performance of the state-of-the-art fuzzer AFL++ by up to 4%. It achieves this despite the additional overhead of regular recompilation, suggesting that any additional scaling to more CPU cores, as would be typical for large fuzzing campaigns with hundreds to thousands of cores, would further increase the performance gains of PogoFuzz. Finally, by focusing on optimizing the target rather than the fuzzer, we have shown that there are other, new opportunities to improve fuzzing.

Acknowledgments

This research is supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972, as well as the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT19056]. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the respective funding agencies.

References

[1] A. Andronidis and C. Cadar. "SnapFuzz: high-throughput fuzzing of network applications." In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2022. DOI: 10.1145/3533767.3534376.

[2] A. Arya and O. Chang. *ClusterFuzz: Fuzzing at Google Scale*. 2019. URL: https://i.blackhat.com/eu-19/Wednesday/eu-19-Arya-ClusterFuzz-Fuzzing-At-Google-Scale.pdf (visited on 07/10/2024).

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain." In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Oct. 2016. DOI: 10.1145/2976749.2978428.

[4] D. Chen, D. X. Li, and T. Moseley. "AutoFDO: automatic feedback-directed optimization for warehouse-scale applications." In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. Feb. 2016. DOI: 10.1145/2854038.2854044.

[5] D. Chen, N. Vachharajani, R. Hundt, X. D. Li, S. Eranian, W. Chen, and W. Zheng. "Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations." In: *IEEE Trans. Computers* 62.2 (2013), pp. 376–389. DOI: 10.1109/TC.2011.233.

[6] M. Christoff. *Chrome just got faster with Profile Guided Optimization*. Aug. 25, 2020. URL: https://blog.chromium.org/2020/08/chrome-just-got-faster-with-profile.html (visited on 07/10/2024).

[7] *Clang Compiler User's Manual - Profile Guided Optimization*. URL: https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization (visited on 11/08/2024).

[8] J. Corbet. *Profile-guided optimization for the kernel*. Sept. 3, 2020. URL: https://lwn.net/Articles/830300/ (visited on 07/10/2024).

[9] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "AFL++: Combining incremental steps of fuzzing research." In: *Proceedings of the 14th USENIX Workshop on Offensive Technologies*. Aug. 2020. DOI: 10.5555/3488877.3488887.

[10] P. Godefroid. *Fuzzing @ Microsoft - A Research Respective*. Apr. 2019. URL: https://patricegodefroid.github.io/public_psfiles/talk-HCSS2019.pdf (visited on 07/10/2024).

[11] Google. *AdressSanitizer*. URL: https://github.com/google/sanitizers/wiki/AddressSanitizer (visited on 07/10/2024).

[12] W. He, H. Yu, L. Wang, and T. Oh. "Revamping Sampling-Based PGO with Context-Sensitivity and Pseudo-instrumentation." In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*. 2024. DOI: 10.1109/CGO57630.2024.10444807.

[13] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung. "HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores." In: *Proceedings of the 10th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Mar. 2012. DOI: 10.1145/2259016.2259030.

[14] *Honggfuzz*. URL: https://github.com/google/honggfuzz (visited on 07/10/2024).

[15] lcamtuf. *american fuzzy lop*. URL: https://lcamtuf.coredump.cx/afl/ (visited on 07/10/2024).

[16] Meta. *Autonmous testing of services at scale*. May 10, 2019. URL: https://engineering.fb.com/2021/10/20/developer-tools/autonomous-testing/ (visited on 07/10/2024).

[17] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. "FuzzBench: An open fuzzer benchmarking platform and service." In: *Proceedings of the 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Aug. 2021. DOI: 10.1145/3468264.3473932.

[18] Microsoft. *Microsoft Security Risk Detection – What's New*. May 10, 2019. URL: https://devblogs.microsoft.com/premier-developer/msrd-whatsnew/#DevOpsFuzzing (visited on 07/10/2024).

[19] *Mozilla Firefox – Build System – Profile Guided Optimization*. URL: https://firefox-source-docs.mozilla.org/build/buildsystem/pgo.html (visited on 07/10/2024).

[20] S. Nagy and M. Hicks. "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing." In: DOI: 10.1109/SP.2019.00069.

[21] *OSS-Fuzz*. URL: https://github.com/google/oss-fuzz (visited on 07/10/2024).

[22] M. Panchenko, R. Auler, B. Nell, and G. Ottoni. "BOLT: A Practical Binary Optimizer for Data Centers and Beyond." In: *Proceedings of the 17th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2019. DOI: 10.1109/CGO.2019.8661201.

[23] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. "VUzzer: Application-aware Evolutionary Fuzzing." In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

[24] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz. "SoK: Prudent Evaluation Practices for Fuzzing." In: *Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP)*. DOI: 10.1109/SP54263.2024.00137.

[25] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types." In: *Proceedings of the*

30th USENIX Security Symposium (USENIX Security). Aug. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo.

[26] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. "NEUZZ: Efficient Fuzzing with Neural Program Smoothing." In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 2019. DOI: 10.1109/SP.2019.00052.

[27] H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li. "Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Mar. 2023. DOI: 10.1145/3575693.3575727.

[28] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. 2016. URL: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf.

[29] *Use profile-guided optimization | Android Open Source Project*. Aug. 26, 2024. URL: https://source.android.com/docs/core/perf/pgo (visited on 11/11/2024).

[30] W. Xu, S. Kashyap, C. Min, and T. Kim. "Designing New Operating Primitives to Improve Fuzzing Performance." In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Oct. 2017. DOI: 10.1145/3133956.3134046.

[31] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu. "Debloating Address Sanitizer." In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen.

[32] Y. Zhang, C. Pang, S. Nagy, X. Chen, and J. Xu. "Profile-guided System Optimizations for Accelerated Greybox Fuzzing." In: *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Nov. 2023. DOI: 10.1145/3576915.3616636.

Appendix A
Revision Requirements

The reviewers request that the authors address the following points for the full paper:

1) The performance improvement should be evaluated against additional fuzzers, e.g. Honggfuzz and LibAFL.
2) The proposed approach should also be compared with related methods, including prior PGO-assisted fuzzing by Zhang et al. [32] and the Untracer technique by Nagy and Hicks [20], among others.