Journal of Science and Technology – Data Science and Artificial Intelligence
Volume 36, Issue 1, October 2026, 072–079

Co-GRFuzz: Shared-corpus Cooperative Fuzzing for Python Libraries

Anh Le Minh Viet[1], Hieu Nguyen Huy[2], Thang Hoang Nam[2], Anh Tran Hai[1], Truong Tran Xuan[3,*]

[1]*Hanoi University of Science and Technology, Ha Noi, Vietnam*
[2]*Hanoi University of Civil Engineering, Ha Noi, Vietnam*
[3]*Penn State Harrisburg, Penn State University, Middletown, PA 17057, United States*
[*]*Corresponding author email: truong.tran@psu.edu*

Abstract
Parallel fuzzing allows workers with distinct mutation strategies to exchange coverage-increasing inputs, but the resulting edge-set complementarity in Python libraries remains insufficiently characterized. We study co-GRFuzz, a corpus-only cooperative framework in which workers share admitted seeds while maintaining independent coverage histories and learning states. Four two-worker configurations anchored by Pythonfuzz—PP (Pythonfuzz+Pythonfuzz), PD (Pythonfuzz+DQN), PO (Pythonfuzz+PPO), and PG (Pythonfuzz+GRPO)—are evaluated on five Python libraries. We compare local and global edge coverage with isolated single-process references using a common `sys.settrace`-based line-transition coverage representation. Outcomes vary by target under these fixed settings and do not identify GRPO as a consistently strong cooperative partner. On `python-dateutil`, PD and PO each obtain a 3407-edge union, versus 2913 for PG, although their learning-guided workers reach nearly identical local totals. On `cryptography`, PD and PO reach 47 union edges, whereas PG remains at 32. We also analyze worker-unique edges and direct-parent seed provenance to distinguish final coverage contribution from observed seed reuse. In a separate provenance-enabled PG campaign, 106 global-first discoveries have a peer-produced seed as their immediate parent; this establishes mutation lineage, not a causal benefit from sharing. Each configuration has a single six-hour campaign, and isolated single-worker references are not equal-resource two-worker controls. Consequently, results are descriptive rather than statistically generalizable. Finally, two previously undocumented exceptions in real libraries provide qualitative bug-finding evidence.

Keywords: collaborative fuzzing, coverage-guided fuzzing, python, reinforcement learning, seed synchronization.

1. Introduction

Python is widely used in web, cloud, data, and artificial-intelligence systems, making the reliability of reusable libraries important. Coverage-guided fuzzing addresses this problem by repeatedly mutating inputs, executing the target, and retaining inputs that expose previously unseen behavior [1–4]. Learning-guided fuzzing adds a second decision layer by estimating where or how to mutate, but the additional inference and online training can change both execution cost and search behavior [5–7].

Parallel and collaborative fuzzers provide another way to diversify search: independently evolving workers can exchange interesting seeds while retaining different mutation logic or internal state. This idea is well established in AFL-style parallel fuzzing and in ensemble systems, so corpus synchronization itself is not the novelty pursued here. The question we study is narrower and matches the available controls: how do local and campaign-level edge sets observed in shared-corpus two-worker runs compare with the same strategies running alone from an isolated corpus, and which target–pair combinations retain non-overlapping edges? The direct-parent provenance analysis further records when a peer-produced seed is the immediate parent of a global-first discovery, without treating that event as a counterfactual measure of causal benefit.

Our earlier work, GRFuzz, uses Group Relative Policy Optimization (GRPO) to learn byte-position priorities online [8]. In this study, GRPO is one of several mutation strategies rather than the center of the evaluation. We use **co-GRFuzz** as a common corpus-only experimental framework supporting Pythonfuzz-, DQN-, PPO-, and GRPO-guided workers. Coverage histories and learning states remain local; only admitted seeds are exchanged.

The study makes three contributions. First, it defines a common Python coverage and corpus harness for mutation-based, DQN-guided, PPO-guided, and GRPO-guided workers, allowing local and union edge sets to be compared in one coverage space. Second, it compares the shared-corpus campaigns with isolated single-process references and augments endpoint coverage with worker-unique edges and direct-parent provenance. Third, it reports a concrete negative result for the GRPO pairing: strong isolated or local coverage does not translate into strong cooperative contribution on several targets, most clearly on `python-dateutil`, where the GRPO worker reaches nearly the same local total as DQN and PPO but retains far fewer edges outside the Pythonfuzz set. The campaign matrix contains one six-hour run per configuration, so these observations are not used for significance claims.

p-ISSN 3093-3331
e-ISSN 3093-3323
https://doi.org/10.51316/jst.194.dsai.2026.36.1.8
Received: Aug 25, 2026; Revised: Oct 10, 2026;
Accepted: Oct 10, 2026; Online: Oct 15, 2026

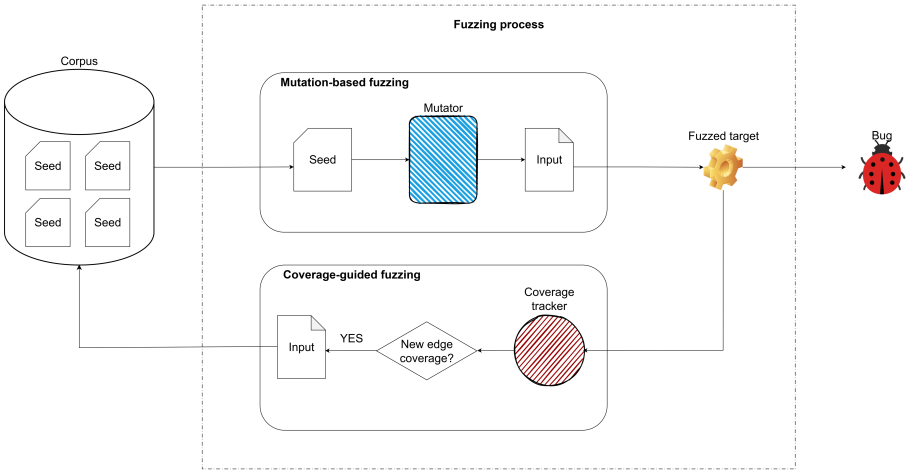


Fig. 1. Mutation-based coverage-guided fuzzing

2. Related Work

2.1. Mutation-Based and Learning-Guided Fuzzing

Mutation-based coverage-guided fuzzing repeatedly selects a seed, mutates it, executes the target, and retains inputs that expose new behavior [2–4, 9] as illustrated in Fig. 1. For compiled programs, coverage is usually obtained through compiler-level instrumentation; Python fuzzers can instead use runtime tracing [10, 11]. Pythonfuzz and Atheris are representative Python-specific fuzzers, although they use different execution and instrumentation paths [12, 13].

Learning-guided fuzzing modifies this loop by predicting promising bytes or mutation actions. Neural Byte Sieve estimates byte-level mutation importance, Deep Reinforcement Fuzzing applies DQN, and Rainfuzz uses PPO-based guidance [5, 7, 14]. GRFuzz applies GRPO to byte-level mutation prioritization and avoids a separate critic network [8, 15]. These approaches differ not only in learning objective but also in inference frequency, update path, and execution cost, which can affect the search trajectory independently of final local coverage.

2.2. Ensemble and Collaborative Fuzzing

Seed exchange and heterogeneous collaboration have been studied extensively. EnFuzz synchronizes seeds among diverse fuzzers so that complementary search strategies can benefit from one another [16]. PAFL adapts single-mode fuzzing optimizations to industrial parallel execution [17]. Cupid selects collaborators automatically, while autofz composes and reallocates fuzzers at runtime based on observed performance [18, 19]. AFL++ integrates many incremental fuzzing techniques in a practical platform, and LibAFL provides reusable components for constructing modular fuzzers and collaborative workflows [20, 21].

Our scope is deliberately narrower than these systems. We do not introduce a new ensemble scheduler, adaptive resource allocator, or distributed coordination algorithm. Instead, we fix the worker count and communication mechanism and use one Python coverage space to compare isolated single-worker outcomes with shared-corpus two-worker outcomes, final edge-set overlap, and direct-parent peer-seed reuse events. The study therefore evaluates mutation-strategy behavior under a fixed cooperation mechanism rather than proposing seed synchronization itself as a new mechanism.

3. Methodology

3.1. Coverage Instrumentation and Cooperation Metrics

All evaluated workers use a custom Python tracer built on `sys.settrace`; `coverage.py` is not used. Before each test input, the worker resets the tracer state, installs the trace callback immediately before calling the target, and disables tracing when the target execution returns or raises an exception. The callback processes only Python `line` events. If the previous executed Python line is (f_s, ℓ_s) and the current event is (f_d, ℓ_d), the coverage unit is the four-field edge

$$e = (f_s, \ell_s, f_d, \ell_d) \tag{1}$$

Here f is the code object's source filename and ℓ is the executed source-line number. The transition is not restricted to one function or one module: a caller-to-callee transition and the first subsequent Python line after a return can therefore cross function or module boundaries. Call and return events are not coverage units by themselves; only consecutive Python line events are recorded. Native C/Rust execution does not emit Python line events, so logic below the Python boundary is outside this coverage space. This limitation is particularly relevant to targets such as `cryptography`.

For one input x, let $\mathscr{E}(x)$ be the set of such line-transition edges. Per-input trace state is reset for every execution, while cumulative campaign coverage is owned by the parent fuzzer process; worker restarts after a timeout therefore do not erase already observed

edges. All Pythonfuzz-, DQN-, PPO-, and GRPO-guided workers record the same four-field edge identity and execute the same target harness, which makes their local sets and offline unions directly comparable within this study. This common edge representation is also why Atheris is not included in the numerical matrix: its libFuzzer-based instrumentation does not emit the same Python line-transition tuples.

For a corpus Q, cumulative coverage is

$$C(Q) = \left| \bigcup_{x \in Q} \mathscr{E}(x) \right| \tag{2}$$

Each worker p maintains its own cumulative edge set $\mathscr{C}_p$. For a multi-worker campaign we report both

$$C_{\max} = \max_p |\mathscr{C}_p| \qquad C_{\text{global}} = \left| \bigcup_p \mathscr{C}_p \right| \tag{3}$$

The union is reconstructed offline and is never fed back into corpus admission, reward computation, or policy updates. For two workers A and B, we additionally report $U_A = |\mathscr{C}_A \setminus \mathscr{C}_B|$, $U_B = |\mathscr{C}_B \setminus \mathscr{C}_A|$, and overlap $O = |\mathscr{C}_A \cap \mathscr{C}_B|$, so that $C_{\text{global}} = U_A + U_B + O$.

3.2. Corpus-Only Cooperative Execution

All workers use one shared corpus but maintain independent execution histories and, where applicable, independent learning state. A fresh campaign starts from the same deterministic one-byte seed, `b"\x00"` (hexadecimal `0x00`), which is inserted when no usable input is present. A worker publishes a mutated input x' when it expands that worker's local coverage:

$$Q_s \leftarrow Q_s \cup \{x'\} \quad \text{if} \quad \mathscr{E}(x') \setminus \mathscr{C}_p \neq \varnothing \tag{4}$$

Seeds are written atomically and deduplicated by content identity. Peers observe new files through filesystem notifications plus a reconciliation scan. Coverage maps, reward buffers, rollout batches, replay state, and policy parameters are not synchronized.

The GRPO-guided worker follows the current GRFuzz implementation. For each non-empty seed, the policy is evaluated over valid byte positions and $G = 64$ positions are sampled with replacement from the frozen behavior policy. Each sampled position mutates a fresh copy of the same seed; rewards are evaluated against one pre-group local-coverage snapshot and normalized into group-relative advantages before the policy update. Full GRFuzz reward and optimization details are described in our earlier work [8]. In the cooperative experiments, these updates remain entirely local to the GRPO worker.

For the provenance case study, each generated seed is identified by a content hash and associated with its first producer. If a peer directly mutates that seed and discovers an edge that is globally new at that time, the event is recorded as a direct-parent peer-seed event attributed to the producer. This records the immediate mutation lineage observed in the campaign; it is not a counterfactual claim that seed sharing caused the edge or that the edge could not later be discovered through another path.

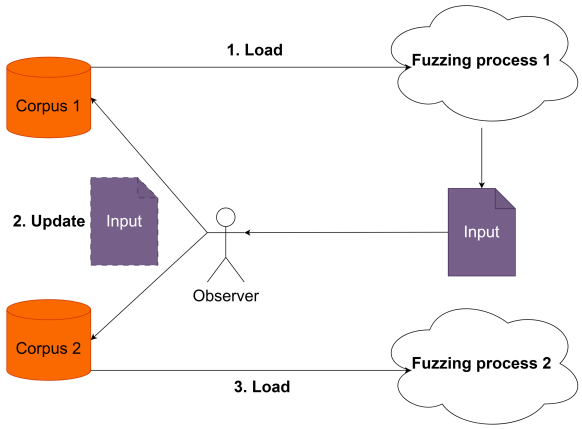


Fig. 2. Corpus-only cooperative execution with independent coverage and learning state

4. Experiments

4.1. Experimental Setup

Experiments were conducted on Ubuntu 24.04.2 LTS (x86_64) with 16 CPU cores and 66 GiB RAM using Python 3.12.2, TensorFlow 2.18.0, `psutil` 6.1.1, and `watchdog` 6.0.0. Each campaign ran for six hours from a fresh corpus initialized with `b"\x00"`. No CPU-affinity pinning was used: neither the experiment launcher nor the worker processes assigned dedicated cores (for example through `taskset` or `sched_setaffinity`); worker placement was left to the Ubuntu scheduler. The same no-pinning policy was used for every pairing, so throughput and RSS are interpreted as descriptive measurements that may include scheduler-placement noise. Five PyPI targets were selected to cover cloud APIs, encoding, date parsing, serialization, and cryptographic bindings (Table 1).

The single-process comparison uses Pythonfuzz [12], PyDQNfuzz [5], PyRainfuzz [7], and GRFuzz [8]. The two-worker matrix fixes Pythonfuzz as one worker and varies the second worker: PP (Pythonfuzz+Pythonfuzz), PD (Pythonfuzz+DQN), PO (Pythonfuzz+PPO), and PG (Pythonfuzz+GRPO). This yields 20 target–pair campaigns. A separate provenance-enabled `python-dateutil` run compares PP, PG, and GG (GRPO+GRPO). The provenance run is analyzed as a mechanism study rather than as a replicate of the main matrix.

We report cumulative edge coverage, execution throughput, and resident set size (RSS). For two-worker campaigns we additionally report $C_{\max}$, C_{global}, and worker-unique edges. Throughput is reconstructed from cumulative execution counters and timestamps; RSS comes from periodic process logs.

Table 1. Target Python libraries and functions under test

Library	Target function(s)	Purpose
botocore	`serialize_to_request()`	Normalizes AWS API request objects.
idna	`encode()`, `decode()`	Encodes and decodes internationalized domain names.
python-dateutil	`decode()`, `tzstr()`	Parses and converts date/time-zone strings.
pyyaml	`safe_load()`	Parses and loads YAML input.
cryptography	`update()`, `sign()`, `verify()`	Exercises hashing and DSA signing/verification APIs.

Table 2. Fixed implementation settings of the learning-guided workers

Method	Input / model	Optimization	Exploration / update settings
DQN	max input 1024 B; 32-byte state; Dense 64–128, tanh; 5 actions	Adam, 10^{-3}; $\gamma = 0.9$	ε: $0.7 \rightarrow 0.1$, decay 0.999/action; replay disabled
PPO	max input 1024 B; Dense 128, tanh; position policy	Adam, 10^{-4}; clip 0.5; entropy 3.0	buffer 50; 75% random positions; policy updates on non-random actions
GRPO	max input 1024 B; 2×Dense 64, swish; position policy	Adam, 10^{-3}; clip 0.1; entropy 3.0	group 64; policy sampling with replacement

Table 2 reports the fixed implementation settings actually used by the three learning-guided workers. These are execution configurations, not results of a common hyperparameter-optimization study. DQN uses a 32-byte state, a 64/128-unit tanh network, five mutation actions, Adam with learning rate 10^{-3}, $\gamma = 0.9$, and an epsilon schedule from 0.7 toward 0.1 with factor 0.999 per action; replay is disabled. PPO uses a 1024-byte input representation, one 128-unit tanh hidden layer, Adam with learning rate 10^{-4}, clip parameter 0.5, entropy coefficient 3.0, a 50-sample update buffer, and 75% random position selection. GRPO uses a 1024-byte representation, two 64-unit swish hidden layers, Adam with learning rate 10^{-3}, group size 64, clip parameter 0.1, entropy coefficient 3.0, and policy sampling with replacement. No cross-method tuning sweep was performed, so algorithm comparisons include the effect of these fixed implementation choices.

Fuzzing is stochastic, and established evaluation guidance recommends repeated trials, standardized benchmarks, and distributional or effect-size analysis rather than conclusions from a single run [22–26]. The present matrix contains one observed campaign per configuration. We therefore report the raw outcomes and within-run edge decomposition without significance testing.

Atheris is discussed as a relevant Python fuzzer but is not included in the numerical matrix because the evaluated workers use the `sys.settrace`-based four-field edge representation defined in Section 3.1, whereas Atheris follows a libFuzzer-based instrumentation path. Its native coverage units therefore cannot be unioned directly with the Python line-transition tuples without an additional harmonization layer.

4.2. Results

4.2.1. Single-process baseline and runtime profile

Table 3 shows that the single-process results are target-dependent. All four fuzzers tie on `botocore` and `cryptography`. PyDQNfuzz is slightly highest on `idna` (1038 edges), while Pythonfuzz leads on `python-dateutil` and `pyyaml`. Among the learning-guided methods, GRFuzz is highest on `python-dateutil` (1762) and `pyyaml` (2132). These data do not support a general claim of coverage superiority for any learning algorithm.

Table 4 separates execution cost from coverage outcome. Pythonfuzz has the highest observed throughput and by far the lowest RSS on all targets. Among the learning-guided implementations, GRFuzz has the highest throughput on four of five targets and the lowest mean RSS on all five. The RSS difference relative to the other learning-guided implementations is modest, however—roughly 1–17% depending on target and comparator—and the measurements do not isolate model memory from runtime overhead. We therefore treat RSS as descriptive system context, not as evidence of a general efficiency advantage. Throughput likewise does not predict coverage monotonically: the much slower DQN is highest on `idna`, while on `python-dateutil` the faster GRPO-guided worker participates in a 2913-edge union, below the 3407-edge unions observed for PD and PO. Execution rate is therefore reported as context rather than as a causal explanation for coverage.

4.2.2. Shared-corpus versus isolated references

The single-process runs provide an isolated-corpus reference for each mutation strategy because only one worker reads and grows the corpus in those campaigns. Table 5 compares each isolated result with the local coverage of the same strategy in the shared-corpus matrix. Because Pythonfuzz appears in every pair, its shared value

Table 3. Single-process edge coverage after six hours

Library	Pythonfuzz	PyDQNfuzz	PyRainfuzz	GRFuzz
botocore	**170**	**170**	**170**	**170**
idna	1031	**1038**	934	936
python-dateutil	**2417**	1539	1750	1762
pyyaml	**2291**	2036	2042	2132
cryptography	**32**	**32**	**32**	**32**

Table 4. Throughput (exec/s) and mean RSS (MiB) by fuzzer and target

Target	Pythonfuzz	PyDQNfuzz	PyRainfuzz	GRFuzz
botocore	**3075.82 / 55.47**	27.33 / 970.59	72.05 / 1012.78	1108.95 / 961.59
cryptography	**252.62 / 52.43**	22.52 / 1118.07	71.52 / 1016.18	48.36 / 960.61
python-dateutil	**2873.04 / 72.16**	26.38 / 1161.67	74.78 / 1016.65	1096.95 / 962.97
idna	**2845.01 / 42.55**	26.06 / 1152.17	67.52 / 1012.61	1063.71 / 951.43
pyyaml	**1497.88 / 45.16**	13.52 / 1051.97	38.26 / 1043.81	856.38 / 956.19

is reported as the range across PP, PD, PO, and PG; DQN, PPO, and GRPO correspond to the second worker in PD, PO, and PG.

The strongest shared-versus-isolated shift appears on `python-dateutil`: Pythonfuzz rises from 2417 to 2906–2907 local edges, DQN from 1539 to 2054, PPO from 1750 to 2054, and GRPO from 1762 to 2055. On `pyyaml`, DQN, PPO, and GRPO are higher in the shared runs, whereas Pythonfuzz is lower (2221–2247 versus 2291). On `idna`, the guided workers change only slightly, while Pythonfuzz varies more strongly across partners (937–1039 versus 1031 in isolation). Neither `botocore` nor `cryptography` changes in local cardinality. At campaign level, PD and PO on `python-dateutil` reach 3407 edges compared with 2417 for the strongest isolated constituent, while PG reaches 2913. Conversely, every shared-corpus pairing on `pyyaml` remains below the isolated Pythonfuzz result of 2291 edges, and PG on `idna` remains below the isolated Pythonfuzz result of 1031.

These comparisons provide an isolated single-worker reference but are not an equal-resource ablation of corpus sharing. A two-worker shared campaign differs from a single-worker run in both seed availability and parallel execution, and each condition is represented by one stochastic campaign. The observed differences can therefore be associated with the shared-corpus setting but cannot be attributed causally to sharing alone. An equal-resource control would require two concurrent workers with separate corpora under the same scheduler and wall-clock budget.

4.2.3. Two-worker coverage across five targets

The clearest set-level contrast appears on `cryptography`. Every worker reaches 32 local edges, yet PD and PO each produce a 47-edge union. PG remains at 32, so the GRPO partner contributes no additional final edge beyond the Pythonfuzz set in that campaign. By contrast, `botocore` ends at the same 170-edge union for every pairing, with no worker-unique edges. The homogeneous PP configuration also ends with complete final overlap on all five targets ($U_1 = U_2 = 0$ in every PP campaign).

The largest absolute gains over $C_{\max}$ occur on `python-dateutil` and `idna`. On `python-dateutil`, PD and PO both reach $C_{\text{global}} = 3407$ from $C_{\max} = 2906$, an increase of 501 edges (17.24%). Their DQN and PPO second workers each reach 2054 local edges and retain 501 edges not present in the Pythonfuzz anchor. The GRPO worker reaches nearly the same local total (2055) but retains only six edges outside its Pythonfuzz peer, producing $C_{\text{global}} = 2913$. This is the strongest negative result for GRPO in the cooperative matrix: its isolated coverage on `python-dateutil` is higher than DQN and PPO (1762 versus 1539 and 1750), yet its shared-corpus contribution is much smaller in the observed pair. On `idna`, PD reaches 1409 edges, PO reaches 1308, and PG reaches only 941; on `cryptography`, PD and PO reach 47 while PG remains at 32.

GRPO is therefore not supported as a generally competitive cooperative partner by these data. Its best shared-corpus result is on `pyyaml`, where PG reaches 2255 edges, but this is only eight edges above PP and still below the isolated Pythonfuzz result of 2291. The data instead show that the ranking of a strategy in isolation need not carry over to its contribution when paired with Pythonfuzz. This conclusion is specific to the evaluated implementations, fixed configurations, targets, and single observed campaigns; it does not establish that GRPO is intrinsically unsuitable for cooperative fuzzing.

4.2.4. Seed-provenance case study on `python-dateutil`

The provenance-enabled case study uses an independent PP/PG/GG campaign on `python-dateutil`. It is intentionally analyzed

Table 5. Local edge coverage in isolated and shared-corpus runs

Target	Pythonfuzz	DQN	PPO	GRPO
botocore	170 → 170	170 → 170	170 → 170	170 → 170
idna	1031 → 937–1039	1038 → 1042	934 → 941	936 → 941
python-dateutil	2417 → 2906–2907	1539 → 2054	1750 → 2054	1762 → 2055
pyyaml	2291 → 2221–2247	2036 → 2215	2042 → 2231	2132 → 2237
cryptography	32 → 32	32 → 32	32 → 32	32 → 32

Table 6. Two-worker edge coverage across five targets

Target	Pair	C_1	C_2	$C_{\max}$	C_{global}	U_1	U_2
botocore	PP	**170**	**170**	**170**	**170**	**0**	**0**
botocore	PD	**170**	**170**	**170**	**170**	**0**	**0**
botocore	PO	**170**	**170**	**170**	**170**	**0**	**0**
botocore	PG	**170**	**170**	**170**	**170**	**0**	**0**
idna	PP	**1039**	1039	1039	1039	0	0
idna	PD	1038	**1042**	**1042**	**1409**	**367**	**371**
idna	PO	937	941	941	1308	**367**	**371**
idna	PG	937	941	941	941	0	4
python-dateutil	PP	2906	**2906**	2906	2906	0	0
python-dateutil	PD	2906	2054	2906	**3407**	**1353**	**501**
python-dateutil	PO	2906	2054	2906	**3407**	**1353**	**501**
python-dateutil	PG	**2907**	2055	**2907**	2913	858	6
pyyaml	PP	**2247**	**2247**	**2247**	2247	0	0
pyyaml	PD	2221	2215	2221	2239	**24**	**18**
pyyaml	PO	2231	2231	2231	2231	0	0
pyyaml	PG	2237	2237	2237	**2255**	18	**18**
cryptography	PP	**32**	**32**	**32**	32	0	0
cryptography	PD	**32**	**32**	**32**	**47**	**15**	**15**
cryptography	PO	**32**	**32**	**32**	**47**	**15**	**15**
cryptography	PG	**32**	**32**	**32**	32	0	0

separately from Table 6. The PG pair reaches $C_{\text{global}} = 2915$ in this run, compared with 2913 for PG in the main matrix, a difference of only two edges (about 0.07%). The close endpoint values are useful as a descriptive consistency check, but they do not constitute a repeatability analysis because each value comes from a single campaign and the provenance run uses additional instrumentation. We therefore use the provenance campaign to study within-run transfer mechanisms rather than as statistical confirmation of the main-matrix endpoint.

Within the provenance run, PG adds only six final edges beyond its strongest local worker and only nine beyond the PP union, so its endpoint advantage is small. The seed history nevertheless records 124 published seeds in PG, compared with 78 for PP and 80 for GG. GRFuzz-produced seeds are the direct parent of mutations associated with 52 later global-first discoveries by Pythonfuzz, and Pythonfuzz-produced seeds are the direct parent for 54 such discoveries by GRFuzz, for 106 recorded direct-parent peer-seed events. Many of these edges later become common to both workers and therefore appear in final overlap rather than in U_1 or U_2.

The normalized rates range from 0.70 to 5.21 global-first edges per 10^6 peer executions. These values are directional and depend jointly on the producer seed and the consumer's mutation strategy. They describe observed mutation lineage inside the PG campaign; without an equal-resource isolated-corpus control, they are not evidence that seed sharing caused 106 additional edges.

4.2.5. Scope and limitations

The broad matrix contains one six-hour campaign per target–pair configuration. Although the two PG campaigns on `python-dateutil` end at closely aligned unions of 2913 and 2915 edges, two observations are not sufficient to estimate variance or establish repeatability. Repeated independent campaigns are required to quantify dispersion, perform statistical tests, and report effect sizes in line with established fuzzing and randomized-algorithm evaluation practice [22–26].

Table 7. Two-worker edge contribution in the provenance-enabled `python-dateutil` case study

Config.	C_1	C_2	$C_{\max}$	C_{global}	U_1	U_2	Overlap
PP	2906	**2906**	2906	2906	0	0	2906
PG	**2909**	2057	**2909**	**2915**	**858**	**6**	2051
GG	1750	1750	1750	1750	0	0	1750

Table 8. Direct-parent seed provenance in the `python-dateutil` case study

Config.	Seed producer	Published	Peer reused	Productive	Transfer global edges	Edges/10^6 peer exec.
PP	Pythonfuzz-1	47	47	**13**	**60**	1.63
PP	Pythonfuzz-2	31	31	10	38	1.58
PG	Pythonfuzz	55	55	8	54	**5.21**
PG	GRFuzz	**69**	**69**	**13**	52	1.52
GG	GRFuzz-1	41	41	3	9	0.70
GG	GRFuzz-2	39	39	9	20	1.66

The single-process runs provide isolated single-worker references, and Section 4.2.2 compares them directly with local coverage in the shared-corpus campaigns. They are not, however, an equal-resource two-worker isolated control. Consequently, the observed shared-versus-isolated differences combine seed availability, parallel execution, and run-to-run stochasticity; they cannot isolate the causal contribution of corpus sharing. The broad matrix also fixes Pythonfuzz as the anchor and uses two workers only. Atheris is absent because its native instrumentation is outside the common Python line-transition coverage space. These constraints limit the claims to the reported configurations and motivate repeated trials, two-worker isolated-corpus ablation, larger worker counts, RL–RL pairings, and harmonized Atheris evaluation.

4.3. Bugs Found in Real Libraries

Experiments with GRFuzz exposed two previously undocumented exceptions in the tested library versions. In `python-dateutil` 2.9.0.post0, calling `tzstr("GMT")` without an explicit offset can leave `stdoffset` as `None`; the subsequent multiplication by -1 raises a `TypeError`. The issue was reported at https://github.com/dateutil/dateutil/issues/1432.

In `pyyaml` 6.0.2, `yaml.safe_load("0b_:")` can reach the binary-integer conversion path with an invalid remainder, causing `int(value[2:], 2)` to raise a `ValueError`. The issue was reported at https://github.com/yaml/pyyaml/issues/868. These findings are qualitative evidence of practical bug-finding capability; they are not used to infer that higher edge coverage necessarily implies higher bug yield.

5. Conclusion

This paper compares isolated single-process references with shared-corpus two-worker Python fuzzing campaigns under one `sys.settrace`-based line-transition coverage representation. The results do not support GRPO as the strongest cooperative partner: on `python-dateutil`, DQN and PPO each retain 501 edges outside the Pythonfuzz set while GRPO retains 6 despite nearly identical local totals; PD and PO also outperform PG on `cryptography` and `idna`. PG is highest among the shared pairs on `pyyaml`, but remains below isolated Pythonfuzz. Thus, the single-process ranking does not carry over consistently to the paired setting.

The provenance-enabled PG run records 106 global-first discoveries whose direct parent was a peer-produced seed, which documents mutation lineage but does not establish a causal sharing gain. The main limitation is replication: each configuration has one six-hour campaign, and the single-process references are not equal-resource two-worker isolated controls. Consequently, the paper reports observed campaign outcomes rather than significance or generalization claims. Repeated campaigns, equal-resource isolated-corpus controls, hyperparameter sensitivity analysis, larger worker pools, and a harmonized Atheris coverage layer are required to determine which effects persist.

Data and Code Availability

The source code and cooperative extensions used in this study are publicly available in the GRFuzz repository at https://github.com/shukuchicoding/co-grfuzz, pinned to commit 9702c1fe13b35aa52f28a9bd09119b04c833ad55.

The evaluated targets are public Python packages listed in Table 1. The artifact includes the custom `sys.settrace`-based tracer, per-worker coverage-event logs, runtime/RSS logs, seed-provenance logs, and the analysis scripts used to reconstruct the reported tables. This repository and commit identifier are used consistently throughout the manuscript and response materials.

References

[1] F. C. G. Bogaerts, N. Ivaki, and J. Fonseca, A taxonomy for Python vulnerabilities, IEEE Open Journal of the Computer Society, vol. 5, pp. 368–379, Jan. 2024. https://doi.org/10.1109/OJCS.2024.3422686

[2] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, The Fuzzing Book, 2019. [Online]. Available: https://www.fuzzingbook.org/. Accessed on: May 6, 2025.

[3] M. Zalewski, American Fuzzy Lop (AFL), 2015. [Online]. Available: https://lcamtuf.coredump.cx/afl/. Accessed on: May 2, 2025.

[4] LLVM Project, libFuzzer: A library for coverage-guided fuzz testing, 2018. [Online]. Available: https://llvm.org/docs/LibFuzzer.html. Accessed on: May 4, 2025.

[5] K. Böttinger, P. Godefroid, and R. Singh, Deep reinforcement fuzzing, in 2018 IEEE Security and Privacy Workshops (SPW), pp. 116–122, Aug. 2018.

[6] Z. Zhang, B. Cui, and C. Chen, Reinforcement learning-based fuzzing technology, in International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, pp. 244–253, Jun. 2020.

[7] L. Binosi, L. Rullo, M. Polino, M. Carminati, and S. Zanero, Rainfuzz: Reinforcement-learning-driven heat maps for boosting coverage-guided fuzzing, in ICPRAM, pp. 39–50, 2023.

[8] V.-A. Le-Minh, H.-A. Tran, H.-H. Nguyen, N.-T. Hoang, and T. X. Tran, GRFuzz: A deep reinforcement learning approach to Python library fuzzing with GRPO, in 2025 IEEE International Conference on Information Reuse and Integration and Data Science (IRI), pp. 13–18, Aug. 2025.

[9] B. P. Miller, L. Fredriksen, and B. So, An empirical study of the reliability of UNIX utilities, Communications of the ACM, vol. 33, iss. 12, pp. 32–44, Dec. 1990, 1990. https://doi.org/10.1145/96267.96279

[10] M. Böhme, V.-T. Pham, and A. Roychoudhury, Coverage-based greybox fuzzing as Markov chain, in Proceedings of the ACM Conference on Computer and Communications Security (CCS), pp. 1032–1043, Oct. 2016.

[11] S. Lukasczyk and G. Fraser, Pynguin: Automated unit test generation for Python, in 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE Companion), pp. 168–172, Jun. 2022.

[12] Y. Pats, Pythonfuzz: Coverage-guided fuzz testing for Python, GitLab, 2020. [Online]. Available: https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/pythonfuzz. Accessed on: Oct. 14, 2026.

[13] Google, Atheris: A coverage-guided, native Python fuzzer, 2020. [Online]. Available: https://github.com/google/atheris. Accessed on: May 13, 2025.

[14] M. Rajpal, W. Blum, and R. Singh, Not all bytes are equal: Neural byte sieve for fuzzing, arXiv, 2017. [Online]. Available: https://arxiv.org/abs/1711.04596. Accessed on: Oct. 14, 2026.

[15] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, and D. Guo, DeepSeekMath: Pushing the limits of mathematical reasoning in open language models, arXiv, 2024. [Online]. Available: https://arxiv.org/abs/2402.03300. Accessed on: Oct. 14, 2026.

[16] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers, in 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, pp. 1967–1983, Aug. 2019.

[17] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, PAFL: Extend fuzzing optimizations of single mode to industrial parallel mode, in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 809–814, Oct. 2018.

[18] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz, Cupid: Automatic fuzzer selection for collaborative fuzzing, in Annual Computer Security Applications Conference (ACSAC 2020), pp. 360–372, Dec. 2020.

[19] Y.-F. Fu, J. Lee, and T. Kim, autofz: Automated fuzzer composition at runtime, in 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, pp. 1901–1918, 2023.

[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, AFL++: Combining incremental steps of fuzzing research, in 14th USENIX Workshop on Offensive Technologies (WOOT 20), Aug. 2020.

[21] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, LibAFL: A framework to build modular and reusable fuzzers, in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 1051–1065, Nov. 2022.

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

[23] J. Metzman, L. Szekeres, L. M. R. Simon, R. T. Sprabery, and A. Arya, FuzzBench: An open fuzzer benchmarking platform and service, in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1393–1403, Aug. 2021.

[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 2024 IEEE Symposium on Security and Privacy (SP), pp. 1974–1993, Sep. 2024.

[25] A. Arcuri and L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in Proceedings of the 33rd International Conference on Software Engineering, pp. 1–10, May 2011.

[26] A. Vargha and H. D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong, Journal of Educational and Behavioral Statistics, vol. 25, iss. 2, pp. 101–132, Jun. 2000 . https://doi.org/10.3102/10769986025002101