International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 2, Mar-Apr 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Applications of Machine Learning in Fuzzer Corpus Minimization: A Systematic Literature Review

Kekuluwala Pathiranage Navodya Lakshani*, Kiruparetnam Jude Myuran**
*(School of Computing, University of Staffordshire, United Kingdom
Email: nlkshani02@gmail.com, ORCID - https://orcid.org/0009-0002-4224-2561)
** (Faculty of Science, Engineering and Computing, Kingston University
55-59 Penrhyn Rd, Kingston upon Thames KT1 2EE, United Kingdom
Email: myuranjude@gmail.com,
ORCID- https://orcid.org/0009-0005-7523-0075)

----------**********----------

Abstract:

Fuzzing is established as the de facto standard for software vulnerability testing; however, its efficacy is largely compromised by "corpus bloat", where redundant inputs degrade the feedback loop. This review aims to analyze the evolution of corpus minimization from the Algorithmic Era to the modern-day Generative Renaissance. This paper demonstrates that while the traditional set-cover algorithms such as afl-cmin, MoonLight and OptiMin provide mathematical efficiency, they suffer from "semantic blindness", discarding the valuable seeds that fail to find immediate coverage gains.

Furthermore, the review paper examines the failure of the early Deep Learning integrations in corpus-minimization, which were stifled by the "Vanishing Gradient" problem and the "Latency Wall". This paper identifies a critical pathology and terms the "Semantic Bottleneck": while Large Language Models (LLMs) generate high-fidelity inputs, the industry still relies on legacy heuristic algorithms for corpus minimization. This creates a bottleneck where the diversity gains of the generative agents are pruned by "dumb" filters. The review concludes that the "ML Gap" that was first identified in 2019 remains unbridged. Future research should focus on Lightweight Supervised Learning to achieve efficient intelligence by aligning semantic awareness with the velocity of modern Continuous Integration (CI) pipelines.

Keywords — **Corpus Minimization, Fuzzing, Greybox Fuzzing, afl-cmin, Large Language Models (LLMs), Supervised Learning, Semantic Bottleneck**

----------**********----------

I. INTRODUCTION

The modern software security assurance depends heavily on fuzzing (also known as fuzz testing), a dynamic analysis method tailored to discover vulnerabilities by subjecting a System Under Test (SUT) to a barrage of corrupted input seeds. While feeding random malformed data to a system is simple, the mechanism of the state-of-the-art mutation-based greybox fuzzing is a complex process which involves feedback loops, stochastic optimization, and coverage instrumentation. To understand the need for corpus minimization, this review first looks at the lifecycle of fuzzing and the pathology of corpus bloat.

In the normal Coverage Guided Fuzzing (CGF) architecture, such as in American Fuzzy Lop (AFL) and LibFuzzer, the initial process begins with validating inputs. The fuzzer would operate in a

continuous loop where it selects a seed from the seed corpus, applies mutation, and executes the SUT with the mutated seed, monitoring the execution for any odd behaviour [1]. Such mutation strategies can be vast, ranging from deterministic bit flipping or arithmetic additions to non-deterministic "havoc" where the data is spliced, deleted, or inserted [2].

Code coverage is the critical aspect of this process as it serves as the fitness function of the evolutionary algorithm. The modern fuzzers track edge coverage within the Control Flow Graph (CFG), which is achieved by compile-time instrumentation which injects Lightweight assembly snippets or logging into basic blocks [1] and updates the shared bitmap that maps the branch transitions to hit counts[2]. If a mutated seed triggers a new edge or alters the hit count of an existing edge (suggesting a new loop interaction), the fuzzer notes the seed as 'interesting' [1]. Such input gets saved in the corpus and becomes a permanent parent seed for future mutations [3].

This mechanism, even though successful, creates a new problem, which is "Corpus bloat". When the fuzzer discovers a new path, the corpus grows monotonically. In fuzzing campaigns such as Google's OSS-Fuzz, a corpus would accumulate tens of thousands of seeds [1]. Theoretically, a larger corpus encodes more program logic, but this introduces severe I/O bottlenecks as fuzzing is an I/O bound activity. The overhead of opening, reading, and closing, when coupled with the synchronization costs between parallel workers, introduced a 2× overhead [1]. In Continuous Integration/Continuous Deployment (CI/CD) pipelines where the budgets are time-constrained, for example, 10 mins per commit, I/O latency is catastrophic[4]. The time spent processing redundant inputs is time diverted from the mutation engine, which reduces the executions per second (exec/s) and the probability of triggering deep-state vulnerabilities. Therefore, the reduction of the corpus size without changing the coverage capability, known as corpus minimization or corpus distillation, becomes critical in Fuzz testing [5].

II. ALGORITHMIC ERA OF CORPUS MINIMIZATION

Between 2019 and 2021, the research has been mainly focused on resolving the corpus bloat problem by formalizing minimization as a discrete combinatorial optimization problem. This time, research has moved away from ad-hoc removal strategies and has progressed towards rigorous mathematical models, such as the Weighted Minimum Set Cover Problem (WMSCP) and Maximum Satisfiability (MaxSAT). This period of time is referred to as "The Algorithmic Era" in this review paper.

A. Moonlight

The tool MoonLight (2019) can be presented as the seminal work of this era, which has formalized the corpus minimization as an instance of WMSCP. The problem defines a universe consisting of all instrumentation data edges that have been observed during the execution of the full corpus [5]. The objective is to select the smallest subset of seeds from the larger corpus such that the union of the coverage in the subset equals the total universe coverage. Even though the unweighted version minimizes the number of seeds, Moonlight has introduced a cost function for each seed. This changes the objectives to minimizing the cumulative cost of the selected set[5].

To achieve the above, Moonlight employs a comparative logic: row and column dominance. This method allows the algorithm to mathematically compare every seed against every other seed. If a specific seed covers the exact path as another seed and is smaller in size or executes faster than the other seed, the latter seed is deemed 'dominated' and is discarded. This way, the Moonlight guarantees the corpus is not only smaller but is tailored with the most efficient seeds available for the unique code paths.

Moonlight relies on a greedy approximation to solve the NP-Complete WMSCP. When optimal reductions like dominance are exhausted, the heuristic algorithm selects the seeds that maximize the ratio of newly covered edges to seed weight, which highlights the cruciality of the weighting criteria. This research has shown that weighting by file size is superior to unweighted minimization[5]. Furthermore, the research highlights that weighting

by execution time directly addresses the throughput bottleneck since the fuzzer can maintain a higher iteration rate by selecting the inputs that execute quickly.

B. OptiMin

OptiMin (2021) pushed the mathematical rigor that was started by Moonlight further. OptiMin encodes the minimization problem into Maximum Satisfiability (MaxSAT). This differs from the greedy algorithms, which might settle for local optima by leveraging the EvalMaxSAT solver to produce a mathematically optimal minimized corpus[1]. OptiMin divides the minimization into two categories: Hard Constraints and Soft Constraints.

Hard Constraints (Coverage) are non-negotiable logical conditions. The solver identifies every unique code edge as a mandatory requirement for a given edge; the solver must pick at least one seed from the group of files that cover it. This logic ensures that the corpus, which was minimized, retains 100% of the original code coverage[1].

Soft Constraints (size/cost) are conditions that represent efficiency goals. The solver aims to reduce the selection count, but each seed is assigned a specific weight that represents its count or file size. When the solver is forced to pick a seed to satisfy the hard constraint, this incurs a penalty that is equal to the assigned weight. The algorithm's objective is to minimize the cumulative penalty of the selected set[1].

This MaxSAT formulation identifies the exact minimum set of seeds that is required to maintain coverage.

C. The Empty Seed

Studies from the same era have revealed a phenomenon called Empty Seed, despite the mathematical accuracy of the MoonLight and OptiMin. Research by Herrera et al. [1] has demonstrated that bootstrapping a fuzzer with a single empty file (or a minimal valid header) sometimes outperforms a corpus that has been carefully minimized[1].

This challenges the assumption that "more initial coverage is always better". The reason is that the aggressive minimization removes the genetic diversity of a corpus. While a minimized corpus is efficient in regression testing (replaying already known paths), it might bias the mutation engine towards specific and complex paths (local optima) that are difficult to mutate further. On the other hand, fuzzing that starts with an empty seed forces the fuzzer to rediscover paths organically, by potentially generating inputs that are structurally simpler to mutate into deep states[1].

III. THE MACHINE LEARNING GAP IN CORPUS MINIMIZATION

Even though the "Algorithmic Era" was successful with the mathematical foundation for corpus minimization in fuzzing through the WMSCP and MaxSAT formulations, the later period from 2019-2023 presents a gap in the usage of Artificial Intelligence (AI) in corpus distillation despite the rapid proliferation of AI in other domains.

In this period of time, Machine Learning (ML) and Deep Learning (DL) have begun to enter the generation and mutation phases of fuzzing, yet corpus distillation remains an untouched area. The industry standard for minimization remains dependent largely on afl-cmin. This section dives into the reasons for this ML gap.

A. The anomaly of Selective Progress

In a comprehensive survey of Machine Learning Applications in Fuzzing, Saavedra et al. [6] have clearly identified this ML gap. After reviewing literature on ML in fuzzing from input generation, symbolic execution, and crash triage, the review has been concluded, stating "To our knowledge, there has not been any research into input minimization or Corpus minimization using ML"[6].

This statement highlights the critical gap in the field of fuzzing. By 2019, it is clear that Neural networks (NNs) and Genetic Algorithms (GAs) have been used

to generate inputs. For example, Long Short- Term Memory (LSTM) networks were used to learn PDF grammar to generate seeds[6], and NEUZZ used Neural smoothing for predicting branching behavior [7]. Yet, minimization remained underexplored by ML.

The minimization was not viewed as a significant bottleneck compared to generation, and the heuristic methods, such as afl-cmin, already achieved 'good enough' results, which creates a higher barrier for ML to enter the field of corpus minimization.[6].

B. Technical Failure 1: Challenges of Binary Sequence Modeling

The primary restriction that prevented corpus minimization from adapting DL for Minimization was the inability of the standard Recurrent Neural Networks (RNN) and LSTMs in processing raw binary formats.

To minimize a corpus beyond simple coverage, a model should have the capability to understand the semantic importance of the file structure by recognizing that a specific four-byte sequence at offset 0x10 is a length field that controls the data chunk at offset 0x4000. Early models that were sequence-based processed data in a sequential manner; when applied to binary files, the sequence length of the input becomes prohibitive.

In the Back Propagation Through Time (BPTT), the gradients that are updating weights based on the long-term dependencies decay as they propagate backward with time steps and layers. This is called a Vanishing Gradient Problem. For a binary file with a size of 1MB, the RNN would need to unroll 10^6 times. As the gradient traverses from the end of the header of the file and approaches zero, it causes the model to 'forget' the structural constraints which were defined at the file header by the time it processes the payload.

The LSTMs were made to mitigate this problem but were limited to sequences of a few hundred to a few thousand tokens. A "token" in Natural Language Processing (NLP) is a sub-word in binary analysis and is often a raw byte. Thus, LSTMs were barely able to comprehend a file header. Research that was done for 'Probabilistic grammar learning' with LSTMs has found that even though LSTMs can learn local syntax, such as closing parentheses, it struggles when producing semantically valid inputs that respect global constraints [6, 8].

Another reason is that these models struggled with the 'blindness to semantics' inherent in raw byte embedding since the binary executables and inputs are high entropy streams. Attempts to reduce dimensionality have been able to make the training feasible in the early stages[7]. It often resulted in lossy representations, which discarded edge-transitions that required precise minimization. These models could not read the files enough to guess which seeds can trigger a bug.

C. Technical Failure 2: Inference Latency and Throughput Bottlenecks

The second reasoning for the ML Gap is the Inference Latency. Fuzzing's efficiency is measured by executions per second (exec/s). The industry tools, such as AFL++, which is written in C/C++, are highly optimized and use fork servers or persistent mode to achieve the throughputs exceeding thousands of executions per second per core[2].

The tool afl-cmin operates on raw bitmap data by using compiled native code. The complexity of the algorithm is normally linear or log-linear with respect to the size of the corpus and when introducing NNs into such a loop creates a bottleneck.

1)Computational cost: one forward pass of deep neural networks involves millions of floating-point operations (FLOPs). Whereas afl-cmin evaluates a seed's utility from fast bitwise operations on a coverage map[2], NNs must perform matrix multiplications.

2)The Inter-Process Communication (IPC) Bottleneck: When using a DL model that uses Python, the fuzzer must solve the gap between the native C runtime and the Python interpreter. This

switch involves serializing coverage data to transfer into the Python process, often through the IPC or shared memory, and waiting for the inference results.

3)The architectural mismatch: Fuzzing is usually CPU-bound while DL is GPU-focused. Supplying every fuzzing node with a GPU for corpus minimization is economically not feasible, and running the NN inference on a CPU further increases the latency gap. DeepGo, A research attempting to integrate DL into fuzzing, has identified this bottleneck, and to mitigate this, DeepGo was forced to train the Virtual Ensemble Environment (VEE) and Reinforcement Learning (RL) models concurrently on a different GPU[7]. Even with this, the research shows that the "wall clock time" includes the overhead of synchronization and training[7].

Finally, the integration of ML into corpus minimization has failed under a negative cost-benefit analysis. Considering a standard fuzzing scenario, if an NN requires 100 milliseconds to predict the usefulness of a seed but the target binary executes the same seed in 1 millisecond, the ML model introduces a 100× latency penalty[8]. For a ML approach to be valid, it needs to reduce the corpus size by a factor proportional to the mentioned slowdown. This latency explains the dominance of afl-cmin in corpus minimization.

D. The Later State of Corpus Minimization

As the review moves from 2023 to modern day, the industry standard for corpus minimization remains largely the same as in the algorithmic era. A review of AFL++ reveals that while the modernized fuzzers support a 'custom mutator API' which allows researchers to use Python plugins, the core of the minimization, which is afl-cmin remains untouched[2].

The advanced fuzzing techniques that were emerging in this period, which used taint analysis and symbolic execution, have used expensive techniques to generate seeds, relying on minimization of traditional methods.

Furthermore, to conclude this section, the review highlights that while ML could theoretically understand why a seed would be important, the speed could not be matched with the existing methods. The 'vanishing gradient' prevented models from understanding complex file formats, and the 'latency wall' prevented ML models from operating at the speed of fuzzing. Up until the emergence of Large Language Models (LLMs), minimization purely remained algorithmic.

IV.GENERATIVE ERA

The Gaps of the early 2020s were defined by the inability of RNNs to process long binary sequences, which were able to overcome by the LLMs in the period of 2024 – 2025. This period is marked as the Generative Renaissance in corpus minimization, where the narrative changes from reductive algorithms to generative agents. In this era, the problem was viewed as a semantic interpretation task where agents use pre-trained models to predict high entropy inputs and code coverage.

A. Rise of ML Agents in Fuzzing

A highlighted feature of the Generative Renaissance is the deployment of Multi-Agent Systems (MAS). Unlike the monolithic fuzzing loops that were used in the Algorithmic era, this era has decomposed generation into specialized roles handled by LLM agents. The best examples are AutoSafeCoder [9] and WhiteFox [10]; fuzzers that use agent frameworks to enhance the discovery of vulnerabilities [9, 10].

B. The Continuity of the Minimization Gap

The integration of LLMs in fuzzing makes a remarkable technological leap, especially in the domain of input generation. Recent literature identifies that tools such as TitanFuzz [11], WhiteFox [10], and G2FUZZ [8] are High-Fidelity Generators that resolve the validity problem that occurred in previous eras. However, a deep analysis of the entire fuzzing pipeline reveals an occurrence of asynchronous evolution: even though the generation technologies of fuzzing have advanced in

the Generative era, the process of corpus minimization remains stuck at the Algorithmic era in 2019, even in 2025 research.

Despite the maturing of the LLM-based generators, the industry standard of corpus minimization has not evolved in parallel. The studies from 2025, such as the work by Wolff et al.[4], when discussing the "shifting fuzzing left" have confirmed that state-of-the-art fuzzers still use afl-cmin for corpus distillation. Thus, the ML Gap identified by Saavedra et al. [6]in 2019 persists.

Furthermore, this review states that reliance on the 2025 era generators feeding into a 2019 era minimizer creates a clear pathology, which this paper terms as the "Semantic Bottleneck", where highly tailored seeds generated by the LLMs are fed into a 'dumb' algorithm that cannot 'understand' the value of the seeds.

The heuristic minimizer afl-cmin operates on the code coverage and views the corpus minimization as a form of "lossy compression"[1], meaning if a semantic feature generated by an LLM would not trigger a new edge in the coverage bitmap, the minimizer deems the seed redundant and deletes it. Thus, the complex semantic inputs crafted by the LLMs are usually discarded by the minimizer due to a lack of semantic resolution to recognize the value of a seed. Also, this leaves us stuck in a paradox where the diversity gains of Generative LLMs are pruned by the 'blindness' of the Algorithmic minimizers[1].

V. CONCLUSION

This review has critically analyzed the journey of corpus minimization in fuzzing, revealing a trap between the "blind efficiency of the 'Algorithmic era' and the unscalable intelligence of the 'Generative era'. While tools such as Moonlight have established a mathematical bound of reduction, the minimizer lacks semantic awareness that prioritizes bug-finding logic. Furthermore, the recent explosion of LLMs has solved the gap of smart generation of inputs yet introduced a "semantic bottleneck" where the industry has the capability to generate smart inputs but still relies on greedy legacy algorithms to distil the corpus.

Consequently, the "ML Gap" that was identified in 2019 remains unbridged. We conclude that the solution lies not in larger models such as LLMs but in efficient intelligence. Future research must move towards Lightweight Supervised Learning to resolve the "Latency Wall" and align the speed of the minimization with the velocity of the modern Continuous Integration (CI) pipelines.

REFERENCES

[1] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, in ISSTA 2021. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 230–243. doi: 10.1145/3460319.3464795.

[2] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Conf. Offensive Technol. (WOOT)*, Aug. 2020, Art. no. 10.

[3] V. J. M. Manès *et al.*, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021, doi: 10.1109/TSE.2019.2946563.

[4] D. J. Wolff, R. Shariffdeen, Y. Noller, and A. Roychoudhury, "Shifting fuzzing left in software workflows," *Empir. Softw. Eng.*, vol. 30, no. 5, p. 146, Sep. 2025, doi: 10.1007/s10664-025-10702-5.

[5] A. Herrera *et al.*, "Corpus Distillation for Effective Fuzzing: A Comparative Evaluation," Sep. 21, 2020, *arXiv*: arXiv:1905.13055. doi: 10.48550/arXiv.1905.13055.

[6] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A Review of

Machine Learning Applications in Fuzzing," Oct. 09, 2019, *arXiv*: arXiv:1906.11133. doi: 10.48550/arXiv.1906.11133.

[7] P. Lin, P. Wang, X. Zhou, W. Xie, G. Zhang, and K. Lu, "DeepGo: Predictive Directed Greybox Fuzzing," Jul. 29, 2025, *arXiv*: arXiv:2507.21952. doi: 10.48550/arXiv.2507.21952.

[8] K. Zhang, Z. Li, D. Wu, S. Wang, and X. Xia, "Low-cost and comprehensive non-textual input fuzzing with LLM-synthesized input generators," in *Proc. 34th USENIX Secur. Symp.*, 2025.

[9] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, "AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing," Nov. 05, 2024, *arXiv*: arXiv:2409.10737. doi: 10.48550/arXiv.2409.10737.

[10] C. Yang *et al.*, "WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models," *Proc ACM Program Lang*, vol. 8, no. OOPSLA2, p. 296:709-296:735, Oct. 2024, doi: 10.1145/3689736.

[11] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, in ISSTA 2023. New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 423–435. doi: 10.1145/3597926.3598067.

