Appears in:
Proceedings of the 33rd IEEE International Conference on Software Analysis, Evolution and Reengineering, Tool Demo Track (SANER Tool Demo 2026)

WasmWeaver: A Framework for Runtime-Aware WebAssembly Program Generation with Reinforcement Learning

Kilian Müller*‡, Siddharth Mane‡, Peter Wägemann†, Norman Franchi‡
*Siemens AG, Foundational Technologies, 91058 Erlangen, Germany
kilian.mueller@siemens.com
†Systems Software Group, Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany
waegemann@cs.fau.de
‡Institute for Smart Electronics and Systems, Friedrich-Alexander-Universität Erlangen-Nürnberg, 91058 Erlangen, Germany
kilian.felix.mueller@fau.de, siddharth.mane@fau.de, norman.franchi@fau.de

***Abstract*—Runtime-aware code generation for sandboxed compile targets such as WebAssembly is scarce yet vital for unbiased training and evaluation of, e.g., large language models (LLMs) tasked with code analysis and schedule optimization. We introduce WASMWEAVER, the first end-to-end generator framework that learns to synthesize fully executable WebAssembly (Wasm) modules while respecting strict limits on execution-trace length and binary size. A Proximal Policy Optimization (PPO) agent guides high-level decisions, and a runtime abstraction, combined with action masks, exposes only semantically valid instructions, preventing undefined behavior and allowing unbounded loops and division without additional checks. Depending on the reward design, we show that: (A) the generator can produce programs with controllable trace lengths and opcode unigrams matching those of real-world Wasm corpora, and (B) when used in an LLM-in-the-loop setting, the generator learns to produce increasingly complex code samples over time to expose reasoning errors. We evaluate WASMWEAVER with two state-of-the-art models, gpt-4o and o3-mini. The results demonstrate that our framework reliably generates pseudo-random programs that satisfy hand-crafted, high-level composite rewards and exposes shortcomings in current LLM code-reasoning/analysis capabilities. The tool's source code and a brief demonstration video are available at github.com/Morxos/wasmweavercode.**

***Index Terms*—Reinforcement learning, Program synthesis, Benchmark testing, WebAssembly, Large language models**

I. INTRODUCTION

Large language models (LLMs) have recently advanced in program generation and execution reasoning [1], enabling applications such as vulnerability detection and runtime analysis. While these models perform well on tasks that require only shallow semantic understanding, their behavior on domain-specific, execution-dependent analyses remains unclear. WebAssembly (Wasm), a compact and increasingly widely deployed compilation target—including in industrial and safety-critical environments–poses a particularly relevant testbed: reasoning about Wasm execution requires precise tracking of stack state, control flow, and resource bounds.

Evaluating such reasoning, however, is difficult. Existing Wasm benchmarks [2] lack detailed execution annotations, making it impossible to obtain reliable ground truth for downstream model assessment. Without such ground truth, comparative benchmarking easily leads to misleading conclusions or data leakage [3]. Automatically generating programs together with their exact execution semantics is therefore essential for any rigorous evaluation of LLMs' reasoning capabilities.

In this paper, we present WASMWEAVER, a framework for generating fully executable, resource-bounded Wasm modules along with precise execution traces. WASMWEAVER couples a tile-based generator with a Proximal Policy Optimization (PPO) agent and a symbolic abstraction of runtime state, enabling controllable program structure, semantic validity, and deterministic labeling.

Our contributions are:

1) A deep reinforcement learning (DRL)-driven, state-aware Wasm generation framework that produces valid modules under configurable bytecode and fuel limits (trace budget), with an extensible tile architecture for new idioms and control-flow patterns.
2) A custom *gymnasium* environment that provides action masking, constraint checking, and multi-objective reward support, facilitating experimentation with different DRL algorithms and curriculum strategies.
3) Standardized task-generation and evaluation harnesses for both reward-driven random program creation and adversarial LLM stress testing.

II. PROBLEM STATEMENT

a) Problem 1: Executability Under Limits: Grammar-aware fuzzers such as `wasm-smith` [4] ensure syntactic correctness but frequently produce programs that trap or fail to exercise meaningful behavior. Recent LLM-based fuzzers [5]–[7] add semantic structure but still ignore strict fuel and byte constraints.

This work was funded by the German Federal Ministry of Research, Technology and Space (BMFTR) project 6G-ANNA (grants 16KISK084 and 16KISK098). This work contributes to the research within the 6G-Valley innovation cluster.

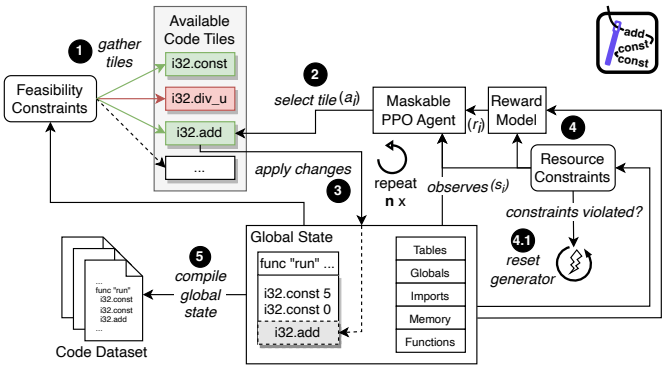


Fig. 1: Overview of the WASMWEAVER framework.

Our solution: WASMWEAVER pairs symbolic execution with a maskable PPO agent that filters infeasible instructions, ensuring that every generated module respects predefined resource limits.

b) Problem 2: Realistic Program Shape: Coverage-oriented generators and corpus splicers [8], [9] distort opcode distributions, control-flow depth, and overall structural complexity compared to real-world Wasm binaries.

Our solution: WASMWEAVER exposes flexible reward functions that steer opcode mix, structural depth, and other high-level metrics toward arbitrary targets, enabling generation of diverse modules that more closely resemble real workloads.

c) Problem 3: Clean, Reproducible Benchmarks: Existing benchmark suites–HumanEval [10], MBPP [11], CodeContests [12]–are small, prone to leakage, and hard to regenerate. LLM-based fuzzers such as GPTFuzzer [13] and Fuzz4ALL [5] lack deterministic ground truth.

Our solution: WASMWEAVER synthesizes an arbitrary number of fully labeled, seed-deterministic Wasm tasks. All samples can be regenerated from configuration files, eliminating contamination and ensuring reproducibility across releases.

III. WASMWEAVER

In this section, we give insight into WASMWEAVER (see Fig. 1). The generator is constructed of two parts. An executor component of WASMWEAVER is able to execute programs step-by-step and updates a global state (symbolic runtime state), tracking all assigned locals, globals, tables, external memory, functions, and the operand stack, as well as their values. Input arguments and memory regions must be known before program generation and are either taken from real-world examples or set randomly. Afterwards, the actual program is generated along the execution flow.

a) Code Tiles & Generation Loop: Code tiles (code fragments) are the smallest unit of code our generator tracks. These can range from simple instructions, through control-flow tiles, up to custom high-level idiom tiles (e.g., recursion and loop patterns). All logic governing whether a tile can be placed at the current point in time, and how it alters the global state when it is placed, is mostly contained within the tile itself, allowing easy extensibility of the generator.

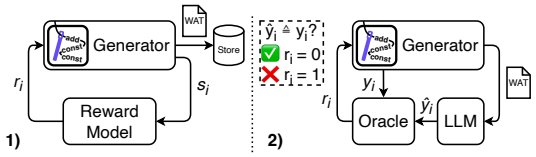


Fig. 2: Two pipelines enabled by WASMWEAVER: (1) reward-driven generation of executable, resource-bounded Wasm with ground-truth traces; (2) LLM-in-the-loop generation that rewards samples where the model's prediction disagrees with ground truth.

The overall generation process, illustrated in Figure 1, repeats the following four steps before outputting the result in step 5: The current state is passed to all available tiles. 1 Each tile responds to whether it can be applied in the current state. 2 A Proximal Policy Optimization (PPO) [14] agent selects a viable action, depending on the current global state and the set of applicable tiles. After selection, the chosen tile is applied 3 to the global state. Constraints are updated and checked. 4 If a constraint is violated, the generator raises an exception 4.1 and the generation process starts over. These four steps are repeated until all constraints are satisfied and the policy finishes the program. 5 If program generation is finished successfully, the resulting global state is compiled into a Wasm binary.

b) Constraints: During generation, we distinguish between two types of constraints: (I) In phase 1, each potential tile evaluates its own local constraints to determine whether it can be placed. These checks include ensuring the stack has the correct shape, verifying that values are within valid ranges (e.g., avoiding division by zero), and confirming that already generated functions can be called. Based on these checks, an action mask (valid-action filter) is generated for the PPO agent to restrict the set of valid actions. These local constraints are designed to reduce the overall search space and prevent the policy from selecting actions that would inevitably lead to invalid program states. (II) After an action is applied, resource and cost constraints are enforced in phase 4. Unlike the local constraints in (I), the status of these global constraints is included in the agent's observation. If any of these constraints are violated, the generator is reset 4.1, and the agent receives a special reward as feedback to encourage learning from the failure.

IV. EVALUATION AND RESULTS

We evaluate two pipelines that illustrate WASMWEAVER's core value (see Fig. 2):

1) Controllable synthesis: generate executable, resource-bounded Wasm programs that satisfy a composite high-level reward under strict fuel/size limits.
2) Ground-truth LLM stress tests: generate labeled, execution-dependent tasks and adversarially steer generation toward cases a target LLM mispredicts. We report results for *gpt-4o* and *o3-mini*.

A. Scenario I: Executable Sample Generation

a) Manual Composite Reward: In many situations, we wish to generate pseudo-random programs that satisfy global constraints. Some of these constraints are *hard*, such as upper bounds on byte-code size or fuel consumption, and are therefore enforced directly during generation (violations simply trigger a generator reset). Others are *soft* preferences, e.g. matching an opcode distribution observed in real-world corpora, steering program length, or encouraging deeper control-flow structure. WASMWEAVER models such preferences through an additional composite reward (sum of objectives).

To illustrate the idea, we assign two distinct rewards and add them up. Our Wasm samples should (a) mimic the opcode distribution of the WasmBench corpus, (b) allow us to dial program length. The first component, the alignment reward R_{align}, is one minus the Jensen–Shannon distance between the opcode histogram produced by the generator and that of WasmBench (see Eq. 1)

$$R_{\text{align}} = 1 - \text{JSD}(P_{\text{gen}} \,\|\, P_{\text{WasmBench}}), \tag{1}$$

where P_{gen} and $P_{\text{WasmBench}}$ denote the respective opcode distributions. The second reward R_{length} measures how closely the actual trace length matches a randomly drawn target fuel value (see Eq. 2).

$$R_{\text{length}} = \exp\Big(-\big|\tfrac{\text{fuel}_{\text{used}}}{\text{fuel}_{\text{target}}} - 1\big|\Big). \tag{2}$$

At inference time this provides a simple knob for generating programs of different lengths.

Finally, the composite reward is the sum of the two parts as in Eq 3.

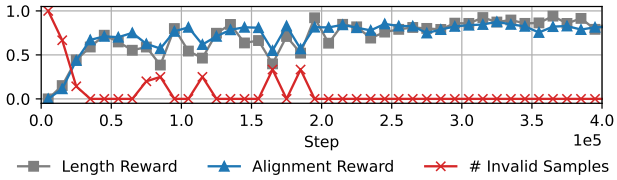
$$R_{\text{total}} = \begin{cases} -1, & \text{failed} \\ R_{\text{align}} + R_{\text{length}}, & \text{otherwise} \end{cases} \tag{3}$$

Should the program generation fail due to a constraint violation, we just return -1 as the total reward.

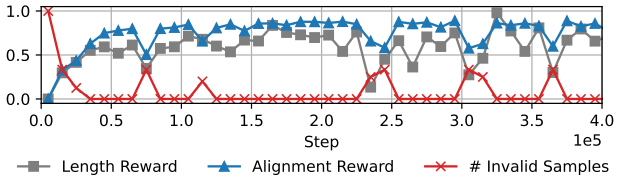
b) Ablation Studies: For understanding how rewards impact program metrics, we perform two leave-one-out ablation studies (one for each reward component) for 400,000 steps each (see Fig. 3b & 3c), with one final study (see Fig. 3a) where all rewards are active. Further, we show three metrics. We observe that in the combined scenario, the generator optimizes both rewards, and once the rewards are turned off, the reward values and their associated metrics decline.

TABLE I: Generalization from Short to Long Fuel Budgets

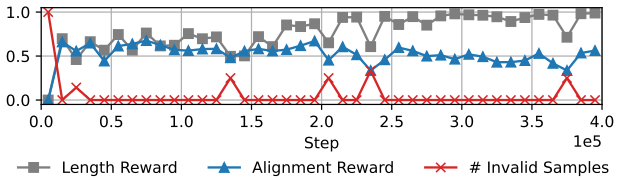
	Max Fuel				
	50	100	200	500	1000
R_{length}	0.87	0.84	0.72	0.67	0.63
R_{align}	0.77	0.75	0.73	0.73	0.73
$\text{MAE}_{\text{length}}$	3.96	6.99	25.88	87.51	207.43
Avg. $\text{Length}_{\text{target}}$	28.31	53.32	102.73	266.69	528.06
Avg. $\text{Length}_{\text{actual}}$	31.20	48.26	76.87	179.71	321.07
Min. $\text{Length}_{\text{actual}}$	7	5	5	4	5
Max. $\text{Length}_{\text{actual}}$	50	99	195	456	824



(a) R_{align} = active; R_{length} = active



(b) R_{align} = active; R_{length} = inactive



(c) R_{align} = inactive; R_{length} = active

Fig. 3: DRL-based program generation with predefined rewards: reward curves/invalid samples over time (a) and ablations with one reward removed (b, c).

c) Generalization to Longer Programs: Due to the target program length input being normalized between 0 and 1 for the generator, we can simply scale up the maximum fuel and binary bounds to observe how this impacts the fuel consumption of generated programs and whether it leads to a collapse in other metrics, such as opcode alignment. We ran this test with maximum fuel values from the initial 50 up to 1000, as shown in Tab. I. Increasing the fuel limit raises the error and lowers the reward for both objectives, yet it does not trigger a catastrophic collapse in any single reward or metric. These results indicate that the approach could also work for scaling more complex composite rewards beyond initial training range for generating longer programs while keeping initial generator training efficient.

B. Scenario II: Adversarial LLM Pipeline

In the second pipeline, we use *gpt-4o* and *o3-mini* in an LLM-in-the-loop setup. The goal here is to find weaknesses in the reasoning capabilities of LLMs used for e.g. scheduling tasks and potential finetuning or training.

a) Stack Reasoning: In the first test, the generator produces simple programs without nested control flow and inserts an *;;INSPECT* comment at a random point. The LLM then lists every value currently on the stack.

Figure 4 shows that in the early phase, when the tasks are simple, both LLMs return correct results. As time passes, the DRL agent shifts its program generation toward workloads that rely more on float conversions. This shift can be observed in

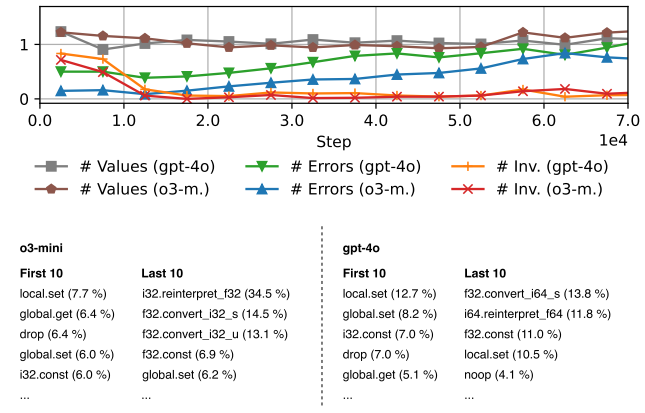


Fig. 4: LLM-in-the-Loop scenario 1: stack-value prediction performance of *o3-mini* vs. *gpt-4o*.

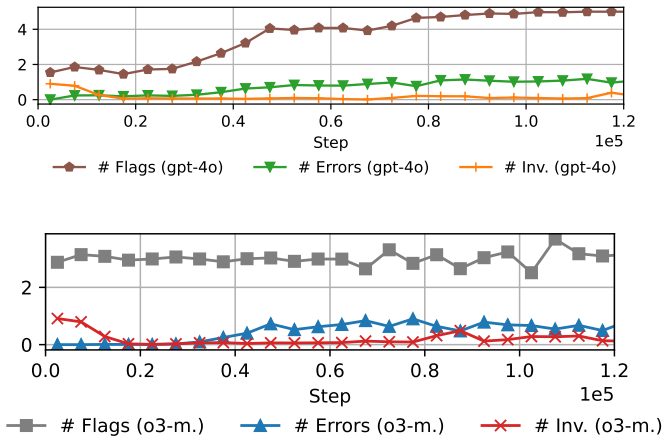


Fig. 5: LLM-in-the-Loop scenario 2: flag-reachability prediction performance of *o3-mini* vs. *gpt-4o*.

the shift of most used opcodes, which can be seen in the lower part of the figure. At first, the opcodes are selected almost at random, but later the distribution concentrates on float-conversion instructions, revealing that both models struggle with conversions between integers and floats.

b) Reachability Reasoning: In the second feasibility study, the DRL agent can generate programs with deeper control-flow structures. After strategic points such as the start of a block or a branch condition it inserts a *FLAG_X* comment. The LLM must decide whether execution will reach the corresponding flag. Because both models struggle with floating-point instructions, we disabled them for this test. Programs use integers only.

As in Fig. 5, a small increase in control-flow depth is often sufficient for the *gpt-4o* model to mispredict about one flag on average. In contrast, for the more advanced *o3-mini* model, the generator converges to programs with complex data-flow chains that end in one or two final decisions. To handle these cases correctly, the model must symbolically execute the preceding instructions in order to determine which flag can be reached. Overall, these results show an average of 0.6 flags being incorrectly predicted per sample.

V. Limitations & Future Work

Section IV shows that WasmWeaver effectively generates random code and benchmarks LLM code reasoning. The study's limitations, however, point to several possibilities for future work.

a) Look-Ahead Design & Lazy Evaluation: At every step, the generator assembles and checks a full list of candidate code tiles. For large constructs such as functions, this exhaustive scan slows generation. In future work, we will replace it with lazy evaluation, thereby deferring the checks for pre-generated tiles until the policy actually selects them. This should markedly accelerate the creation of longer programs with complex control flow.

b) Advanced Control-Flow Patterns: The current prototype of the generator covers only fundamentals–simple loops, blocks, and direct or indirect calls, with the possibility to add more complex idioms with our tiling system. We aim to add richer templates (e.g., complex parametrized loop headers, structured recursion) that lock in the outer skeleton while letting the generator freely fill the bodies.

c) Hierarchical Control Flows: The current version of the WebAssembly specification only supports single-threaded execution; configurations with multiple threads are a subject of future work [15]. In line with the current capabilities of Wasm, our WasmWeaver prototype supports hierarchical control flows; however, we consider the generation of multi-threaded Wasm code with non-hierarchical control flows as relevant future work. One approach in this direction is the use of existing taskset generators, for example, from the domain of real-time systems and schedulability analyses [16], [17], for the automatic generation of multi-threaded Wasm workloads.

VI. Related Work

Below, we discuss related program generators, automatic baselines, and prior work on execution reasoning.

a) Program Code Generators: Yang et al. introduced Csmith, a seminal generator for stress-testing C compilers [18]. For WebAssembly, projects such as WasmCFuzz [19] and wasm-smith [4] output challenging corner cases that expose compiler faults. WasmWeaver follows the idea of structured generation yet pursues a different goal. In Scenario 1, the DRL agent explores the instruction space while symbolic guards keep every program within fuel and byte limits. Its reward scheme is configurable, so researchers can align opcode mix, control-flow depth, or data shapes with any target distribution. Fuzzers like GPTFuzzer [13] and Fuzz4ALL [5] apply LLMs to make inputs more semantic. They generate runnable tasks, but they do not establish a deterministic truth for later evaluation. WasmWeaver instead logs every reachable path and output value, which provides the reference needed for fair comparison.

b) Building Baselines through Generation: GenE produces LLVM intermediate representation for worst-case execution-time studies [20]. GenE and WASMWEAVER share the challenge of creating benchmarks with a known ground truth, though their domains differ. Magma inserts synthetic bugs into hand-picked programs to measure fuzzer coverage [21]. WASMWEAVER adopts the same baseline philosophy but lifts it to autogenerated Wasm tasks that are both diverse and fully labeled.

c) Evaluating LLMs for Execution Reasoning: Liu et al. benchmarked several pretrained models on the CodeNetMut dataset and highlighted contamination issues that arise from public training data [22]. Lyu et al. carried out an exploratory study in which LLMs execute code and return results [23]. WASMWEAVER shares their objective of measuring execution reasoning but removes contamination by generating every task on demand. Use case 2 of WASMWEAVER is unique. After each program run, a judge-LLM inspects the ground truth and its own prediction. Whenever the two differ, the generator gains a positive reward. This feedback loop steers the DRL policy toward tasks that expose blind spots of the judge model, thereby enabling targeted stress tests for future LLM releases.

VII. CONCLUSION

WASMWEAVER, with its ability to produce annotated code and arguments that execute under predefined upper bounds, provides, among other things, the missing ground truth for assessing LLMs on downstream code-reasoning tasks. A mask-guided PPO agent drives a tile-based generator that emits valid, fuel-bounded Wasm and logs every reachable state, removing costly manual annotations and avoiding undecidability traps. Experiments confirm that it can steer opcode mix and length while reliably exposing systematic errors in models such as *gpt-4o* and *o3-mini* in an LLM-in-the-loop configuration. WASMWEAVER thus offers a compact, reproducible framework for code generation and rigorous LLM benchmarks.

AUTHOR CONTRIBUTIONS

K.M.: Conceptualization; Methodology; Software; Investigation; Validation; Writing – original draft. S.M.: Software (initial prototype); Writing – review & editing. P.W. and N.F.: Supervision; Writing – review & editing. All authors reviewed the manuscript and approved the final version.

REFERENCES

[1] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of NLP and software engineering: A survey on language models for code," *Transactions on Machine Learning Research*, 2024. [Online]. Available: https://openreview.net/forum?id=hkNnGqZnpa

[2] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021 (WWW '21)*, 2021, pp. 2696–2708.

[3] E. van der Kouwe, G. Heiser, D. Andriesse, H. Bos, and C. Giuffrida, "Sok: Benchmarking flaws in systems security," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P '19')*, 2019, pp. 310–325.

[4] Bytecode Alliance. (2025) wasm-smith – a webassembly test case generator. [Online]. Available: https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith

[5] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[6] L. Yang, J. Yang, C. Wei, G. Niu, G. Zhang, Y. Wang, L. ChaI, W. Xia, H. Guo, S. Zhang, J. Liu, Y. Yin, J. Peng, J. Ma, L. Sun, and Z. Li, "Fuzzcoder: Byte-level fuzzing test via large language model," 2024. [Online]. Available: https://arxiv.org/abs/2409.01944

[7] H. Zhang, Y. Rong, Y. He, and H. Chen, "Llamafuzz: Large language model enhanced greybox fuzzing," 2024. [Online]. Available: https://arxiv.org/abs/2406.07714

[8] S. Cao, N. He, X. She, Y. Zhang, M. Zhang, and H. Wang, "Wasmaker: Differential testing of webassembly runtimes via semantic-aware binary generation," in *Proceedings of the 33rd ACM International Symposium on Software Testing and Analysis (ISSTA '24)*, 2024, pp. 1262–1273.

[9] J. Han, Z. Zhang, Y. Du, W. Wang, and X. Chen, "Esfuzzer: An efficient way to fuzz webassembly interpreter," *Electronics*, vol. 13, no. 8, 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/8/1498

[10] M. Chen *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[11] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[12] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, p. 1092–1097, Dec. 2022. [Online]. Available: http://dx.doi.org/10.1126/science.abq1158

[13] J. Yu, X. Lin, Z. Yu, and X. Xing, "Gptfuzzer: Red teaming large language models with auto-generated jailbreak prompts," 2024. [Online]. Available: https://arxiv.org/abs/2309.10253

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[15] "WebAssembly Core Specification." [Online]. Available: https://webassembly.github.io/spec/versions/core/WebAssembly-3.0-draft.pdf

[16] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[17] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. of WATERS '10*, 2010, pp. 6–11.

[18] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd Conference on Programming Language Design and Implementation (PLDI '11)*, 2011, pp. 283–294.

[19] X. Zhang, J. Wang, X. Du, and S. Liu, "Wasmcfuzz: Structure-aware fuzzing for wasm compilers," in *Proceedings of the 2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability*, ser. EnCyCriS/SVM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1–5. [Online]. Available: https://doi.org/10.1145/3643662.3643959

[20] P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat, "Benchmark generation for timing analysis," in *Proceedings of the 23rd Real-Time and Embedded Technology and Applications Symposium (RTAS '17)*, 2017, pp. 319–330.

[21] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428334

[22] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan, "Code execution with pre-trained language models," in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 4984–4999. [Online]. Available: https://aclanthology.org/2023.findings-acl.308/

[23] C. Lyu, L. Yan, R. Xing, W. Li, Y. Samih, T. Ji, and L. Wang, "Large language models as code executors: An exploratory study," 2024. [Online]. Available: https://arxiv.org/abs/2410.06667