# Large Language Models for Fuzzing Parsers (Registered Report)

Joshua Ackerman
Dartmouth College
Hanover, NH, USA
joshua.m.ackerman.gr@dartmouth.edu

George Cybenko
Dartmouth College
Hanover, NH, USA
gvc@dartmouth.edu

## ABSTRACT

Ambiguity in format specifications is a significant source of software vulnerabilities. In this paper, we propose a natural language processing (NLP) driven approach that implicitly leverages the ambiguity of format specifications to generate instances of a format for fuzzing. We employ a large language model (LLM) to recursively examine a natural language format specification to generate instances from the specification for use as strong seed examples to a mutation fuzzer. Preliminary experiments show that our method outperforms a basic mutation fuzzer, and is capable of synthesizing examples from novel handwritten formats.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**.

## KEYWORDS

Fuzzing, Parsers, Large Language Models, Deep Learning

**ACM Reference Format:**
Joshua Ackerman and George Cybenko. 2023. Large Language Models for Fuzzing Parsers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop (FUZZING '23), July 17, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605157.3605173

## 1 INTRODUCTION

Faithful input parsing plays an integral role in the security of software. The correctness of many programs rests on the assumption that inputs adhere to some standard format. As such, the violation of this assumption can induce unexpected behavior that can compromise the integrity of the program. Such parser bugs appear frequently, and underlie multiple famous vulnerabilities and exploits such as Heartbleed [36], Psychic Paper [46] and the FORCEDENTRY [33] exploit *inter alia* [4].

Despite the importance of input parsing, developing a correct parser is non-trivial pursuit. One reason for this difficulty is the inherent ambiguity in the very format specifications developers need to implement correctly. Such format specifications are often described in natural language, which subsequently can introduce ambiguity into the specification. The process of disambiguating the specification, is naturally subject to the cognitive biases of the developer, and prior work has shown that common cognitive biases predispose developers to writing insecure code [47]. In particular, developers are known to employ inexact mental heuristics and are prone to cognitive biases [38] that can limit their ability to fully contextualize and anticipate the implications of different decisions such as particular disambiguations of a component of a specification. As such, developers can easily and naturally, make inaccurate assumptions that result in flawed a parser implementation which fails to handle the space of possible inputs correctly.

Given the inherent relationship between ambiguity in the specification and bugs, we propose a method that leverages the natural language format specification for fuzzing parsers. As many prior works has discussed [39] [43] *inter alia*, large language models (LLMs) have issues with bias and inaccuracy. Like developers, they write insecure code [40] and can fail to reason over ambiguity in tasks [28]. We posit these flaws will lead to useful misinterpretations of format specifications that, when used in tandem with a traditional fuzzing algorithm, will outperform similarly black-box fuzzers in parser coverage.

Our method involves using an LLM in combination with an embedding model to recursively examine a natural language format specification to enable the language model to generate instances from the specification that qualitatively align with it. As most format specifications of interest are too large to fit inside of the context window of any LLM, we leverage a secondary embedding model to retrieve relevant sections of the file specification so that the LLM has information to reason over a particular construct in the specification. In our baseline method, the LLM receives questions of the form "What is the structure of a well-formed `object`?" and "Generate an example of an `object`." We pose these questions to an LLM, and recursively query the LLM with questions about the responses until a specified depth, when the model must generate an example of a requested object[1]. This generation process is repeated multiple times to get numerous different examples of each object, which are then recursively combined by the LLM to ideally make well-formed instances of the format. We further query the LLM to mutate these instances to result in errors mentioned by the format-specification.

In this preliminary report, we show that our LLM based method produces strong seeds that when used with a mutation fuzzer on simple targets, exceed the performance of a traditional mutation fuzzer seeded with real examples. We conclude by discussing a more rigorous evaluation and better benchmarks for phase II.


*FUZZING '23, July 17, 2023, Seattle, WA, USA*

ACM ISBN 979-8-4007-0247-1/23/07…$15.00
https://doi.org/10.1145/3605157.3605173

[1] Ideally this object will be a primitive that exists in common knowledge like an integer, or a string. However, in our baseline approach we only allow a fixed number of questions.

# 2 BACKGROUND

## 2.1 Fuzzing

The goal of fuzzing is to test the robustness of a system by identifying vulnerabilities caused by malformed input. A fuzzer repeatedly generates random or malformed inputs, which are fed into the target program. Meanwhile, the behavior of the target program is then monitored for crashes or other unexpected errors. Frequently, coverage of an input i.e., the amount of the target program that is executed as it is evaluated on the input, is also tracked. While there is a wide range of fuzzing techniques (see section 3.1), in this paper, we primarily focus on mutation based fuzzing. Mutation fuzzers often start with an initial *population* consisting of a single*seed* example. The seed example is randomly mutated via a series of transformations such as flipping bits or changing characters, and added to the population. The intent is that as the population grows, so does the total coverage of the program.

## 2.2 Large Language Models

Large language models such as GPT-4 [39] are trained in an *autoregressive* fashion. That is, given a sequence $\mathbf{x} = x_1 \cdots x_t$, they are trained to maximize the likelihood

$$\max_{\theta} \log p_\theta(\mathbf{x}) = \sum_{i=0}^{t} p_\theta(x_i|\mathbf{x}_{<i}).$$

Put simply, they are trained to predict the next word from a given text prefix. While seemingly mundane, the choice of prefix can unlock powerful new capabilities and steer the model to perform tasks it was not explicitly trained to do [8]. This process is called *prompt engineering* and is effective enough that some researchers have begun to call these models *foundation models* [7], due to their versatility that is significantly enabled by prompting.

It is also important to state the importance of *sampling* on the quality of model output. While there are a range of heuristics practitioners employ to sample high-qualtiy results from autoregressive models, in this paper we only concern ourselves with the notion of *temperature*. During training, large auto-regressive models learn a rich hidden representation $h_\theta$, which is then converted to the probability distribution $p_\theta(\cdot)$ via the softmax function,

$$\sigma(h_\theta(\mathbf{x}))_i = \frac{\exp\{h(\mathbf{x}_i)\}}{\sum_{j=1}^{t} \exp\{h(\mathbf{x}_j)\}} \text{for } i = 1, ..., t.$$

These probabilities are often scaled using a temperature term $T$, $\sigma(h_\theta(\mathbf{x})/T)$ to control the sampling behavior. Note that a low value of $T$ exaggerates the probabilities making likely tokens more likely. Likewise, increasing $T$ distributes probability mass more fairly across different tokens. In practice, higher values of $T$ enable more diversity in the output, but increases the chance of a sample being nonsense. For a detailed treatment of sampling in neural text models see [24]. This parameter will be critical in our work, as our fuzzing method needs to strike the right balance between randomness and structure.

# 3 RELATED WORK

## 3.1 Fuzzing

Fuzzing methods are naturally classified by their threat model i.e., the amount of information the fuzzer has access to. *Black-box* fuzzing methods like Radamsa [5], Trinity [26], and, *fuzz* [35] do not assume any information about their target, while *grey-box* methods like LibFuzzer [15], QuickFuzz [20], Superion [50], PeachFuzzer [13] MemFuzz [12] and Angora [10] assume some dynamic access to the target program, such as, being able to monitor its behavior. *White-box* [16] [17] [41] fuzzing methods assume full access to the target program's source code and frequently employ static analysis tools like constraint solvers. Our method fits most intuitively in the black-box, or gray-box domain, as the input generation of white-box fuzzers would directly use program knowledge such as control flow constraints in input generation. Meanwhile, we only assume access to a format specification. While, in this sense our method is black-box, access to additional program information like coverage can be used to increase the performance of our method e.g., by discarding strings in the population that do not lead to increase coverage after some number of mutations.

The closest related fuzzing methods to ours use deep learning in some capacity. In [30], Lemieux et al. use Codex [9] to generate test cases for under-covered functions in search-based software testing. She et al. [44], use neural networks to create a smooth approximation of a program's behavior to enable a gradient guided-approach to input generation. Nichols et al. [37], use a generative adversarial neural network [19] to improve AFL [52], while Böhme [6] use insights generated by a markov chain to improve AFL. Rajpal et al., [42] use neural models like LSTMs to learn from successful mutations. Finally, both LearnFuzz [18] and Skyfire [49] use different models, probabilistic context free grammars and neural networks respectively, to learn well-formed inputs from collections of input samples. One major difference between our method and most of these neural approaches is that our method does not try to model the fuzzing process or the program. Our method also does not require any training and can be used with any pre-trained large-language model.

## 3.2 Semantic Parsing

An area of work conceptually linked to ours, is that of semantic parsing. The overarching goal of semantic parsing is to translate a natural language text into meaningful formal representations. The problem setup in our work is fundamentally different, as the definition of a 'meaningful' input is much looser in fuzzing. In particular, the quality of the result of our fuzzing method is related to, but not dependent on any notion of "accuracy", as slightly malformed inputs maybe beneficial for fuzzing. Nonetheless, there are LLM-based semantic parsing methods that share some ideas with our work. Mekala et al., [34] also use a recursive prompting strategy to ours that guides the language model in decomposing a complex semantic parsing problem into smaller components via series of questions and answering. Shin et al. [45], constrain an LLM to rewrite inputs into an English like 'sub-language' which can be mapped to a formal representation. Like Shin et al., we heavily rely on an LLM's ability to summarize text, however we do not take any action to constrain the output of the LLM.

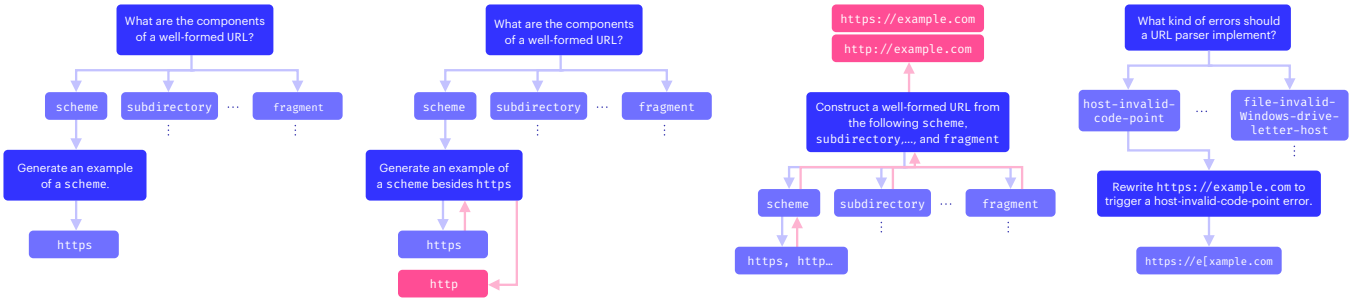


**Figure 1: The four steps of our prompting method: understand, expand, form and deform. This diagram is an illustration – key details such as the inclusion of the file-specification into the prompt are omitted for clarity.**

### 3.3 Program Synthesis

Our work also overlaps with the task of program synthesis from natural language. Traditionally program synthesis methods work from formal or logical constraints [21], but recent advancements in natural language processing have lead to accurate methods of programs synthesis from text. Lei et al. [29], use a Bayesian generative model to map natural language specifications into parsers, however they do not explore any applications of their method to fuzzing, or security in general. In [25] Iyer et al., use a network based on LSTMs [23] to map natural language to source code. Finally, there has been a recent surge of LLM based methods such as GPT-4 [39], GPT-3 [8], Codex [9], PyMT5 [11], and CodeBERT [14] that demonstrate exceptional program synthesis capabilities. Our work actively builds on these advances in natural language processing, albeit for fuzzing programs instead of synthesizing them.

## 4 OUR METHOD

We assume access to a natural language format-specification, $\mathcal{D}$, describing some format that a target parser $P$ implements. We further assume access to a natural language embedding model $\mathsf{Embed}(\cdot)$, and a large-language model $\mathsf{LLM}(\cdot)$ with a context window of size $M$. Our method can be divided into three main stages: (i) ***preprocessing***; (ii) ***seed generation***; and (iii) ***fuzzing***. In the ***preprocessing stage***, we subdivide $\mathcal{D}$ into chunks $\{D_1, D_2, ..., D_n\}$, $|D_i| < M$[2] and embed each chunk section as $E_i = \mathsf{Embed}(D_i)$ to form the set $\mathcal{F} := \{(E_1, D_1), (E_2, D_2), ..., (E_n, D_n)\}$.

For ***seed generation***, we use a novel recursive prompting method that consists of four steps: (i) *understand*; (ii) *expand*; (iii) *form*; and (iv) *deform*. At each step, we augment the prompt with $k$ pieces of relevant information from the format specification [32]. In the *understand* step, we build a parse tree sketch, by querying the LLM with questions about the structure of the file. This process starts by querying the language model with a question $q_0$ of the form "What is the structure of a well-formed `[format-name]`?". We then parse the model's response to extract the different components of its answer that ideally represent high-level parts of `[format-name]`. From here, this process is continued recursively until a pre-specified recursion maximum depth $d_{\max}$ is reached. At this point the model is queried to "Generate an example of `[object]`." Ideally, `[object]` will be a terminal symbol e.g., not composed of other objects, meaning the language model will be able to create a specific instance of it. In the *expand* step, the specific instances are fed back into the prompt, and the language model is asked $\ell$ times to generate new examples of these terminal symbols or return `None` if there are none. Next, in the *form* step, we backtrack up the parse tree sketch and ask the language model combine the generated terminal symbols, ideally resulting in a single well-formed instances of the format. This is repeated for each each of the $\ell$ different terminal symbols, yielding $\ell' \leq \ell$ different examples of the format. During the *deform* step, the language model is asked about the errors $P$ should implement according to $\mathcal{D}$. If the specification mentions such errors, the LLM is asked to mutate each of the $\ell'$ instances to trigger this error.

We now define our method more formally. A fundamental primitive in our method is a format-specification augmented query or *augmented query*. For a question $q = Q(\cdot)$, an *augmented query* involves computing the embedding of $q$, $E_q$, and retrieving the text blocks associated with the $k$ closest embeddings in $\mathcal{F}$ by cosine similarity. That is, we find the set, $\{T_i^* : i = 1, ..., k\}$ where

$$F^* = \{T_i^*, E_i^* : i = 1, ..., k\} = \min_{F \subset \mathcal{F} : |F| = k} \sum_{E_f \in F} \frac{E_q \cdot E_f}{\| E_q \| \| E_f \|},$$

and greedily add each $T_i^*$ to the prompt $p$, ensuring it does not exceed $M$ tokens. The final prompt for the LLM, $\mathsf{LLM}(\cdot)$, is given by $p := T_1^* \oplus \cdots T_{k'}^* \oplus q$, where $\oplus$ denotes string concatenation. We denote an augmented query to an LLM as $\mathsf{LLM}^*(\cdot)$, and a usual query as $\mathsf{LLM}(\cdot)$.

For the *understand* step, we define a question-template function $Q(\cdot)$ that given a string $s$ returns a string similar to "What is the structure of a well-formed $s$?"; and $G(\cdot)$ a generate-template function that takes a string $s$ and a list $L$ returns a string like "Generate an example of $s \notin L$". The exact prompts can be found in the appendix. The understand step begins by making an augmented query to the model with the prompt $Q($`[format-name]`$)$. The model then returns a response $r_0$ which we parse into $r_0^{(0)} r_0^{(1)} \cdots r_0^{(m)}$ to generate new questions $Q\left(r_0^{(0)}\right), ..., Q\left(r_0^{(m)}\right)$. This process is repeated recursively $d_{\max}$ times, at which point the LLM receives augmented queries to generate examples, $G\left(r_{d_{\max-1}}^{(0)}, []\right), ..., G\left(r_{d_{\max-1}}^{(m')}, []\right)$ and produces concrete examples, $r_{d_{\max}}^{(i)}$. The end result of this, is a tree

[2]We found subdividing each format specification by subsections to be effective.

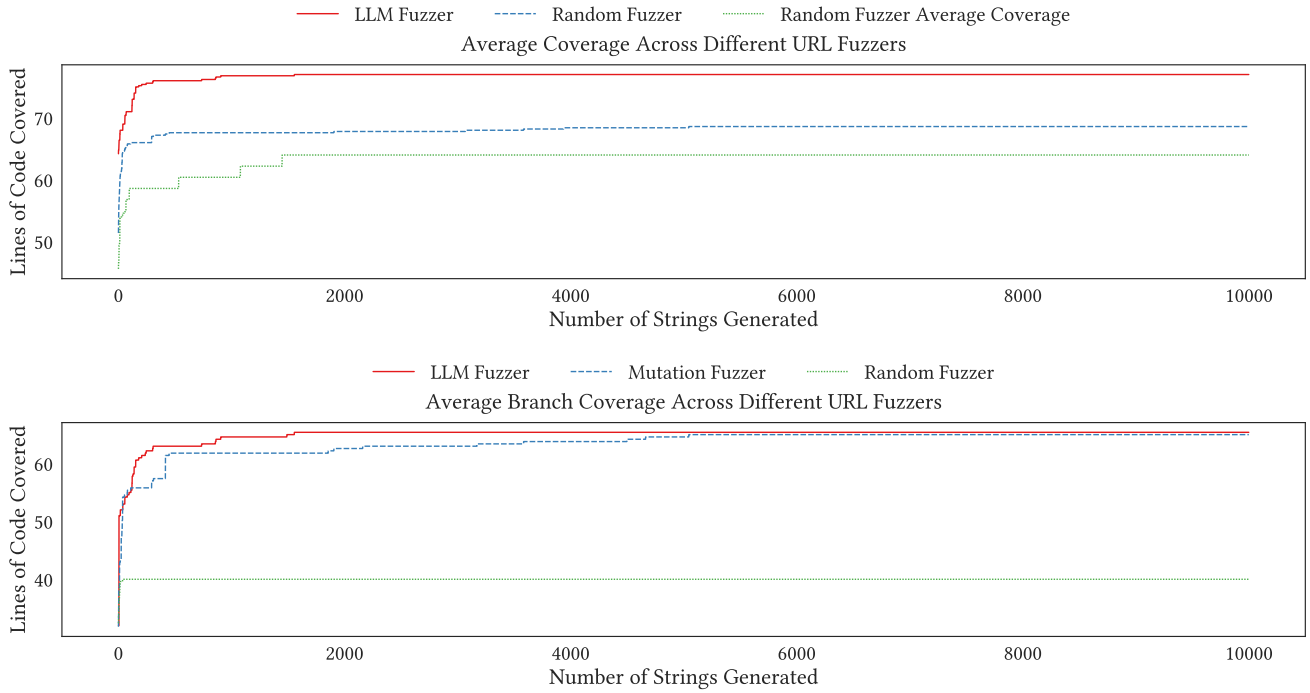


**Figure 2: Performance of all three fuzzers in terms of raw coverage and branch coverage, averaged across ten trials.**

$\mathcal{T}$ that ideally corresponds to something akin to a parse tree of an example. In our implementation we explicitly maintain these responses as an $m$-ary tree, where each node stores a descriptive title $\tau_i$, and a possibly empty set of examples $\{o_i\}_i$.

During the *expand* step, each $r^{(i)}_{d_{\max}}$ is expanded to a set of different examples

$$\left\{ r^{(i)}_{d_{\max},j} = \text{LLM}^*\left(G\left(r^{(i)}_{d_{\max}-1}, \left[r^{(i)}_{d_{\max},j'}\right]_{j'<j}\right)\right), j = 1, ..., \ell' \right\}$$

by maintaining a list of generated responses, feeding them back to the LLM, and prompting it to generate different examples via a sequence of $\ell' \leq \ell$ augmented generation queries. Next, for the *form* step, we define a compose template function $C(\cdot)$ that takes a parent node with title $\tau_p$, and a set of children nodes and outputs a string of the form "Generate a well-formed instance of $\tau_p$ given a $\tau_1, o_1$,..., and a $\tau_n, o_n$." We back-track up the tree, using augmented prompts to the LLM to compose the children of each parent node, to eventually get a set of (ideally) well-formed instances $S = \{s_1, ..., s_{\ell'}\}$. Finally, in the *deform* step, we prompt the model with a general question of the form "What errors should a `[format-name]` parser" implement?" We loop over the errors in the response $r = e^{(0)}e^{(1)} \cdots e^{(m)}$, and send augmented queries to the LLM to rewrite each string in $S$ to trigger each error. The final set of seeds is taken to be the union of this mutated set $S'$, and $S$.

## 5 PRELIMINARY EVALUATION

We structure our evaluation around the following research questions:

- **RQ1**: Does our method get better program coverage than alternative fuzzing approaches?
- **RQ2**: How well does our method generalize to novel formats?
- **RQ3**: Can our method induce targeted errors in programs?
- **RQ4**: Can our method produce diverse, well-formed instances from a file specification[3]?
- **RQ5**: How sensitive is the success of our method to the choice of LLM and accompanying hyper-parameters?

In our phase I evaluation, we present initial results for **RQ1** and **RQ2** on simple tasks as a proof-of-validity of our method. In our preliminary experiments, we wish to demonstrate two important points: (i) our method can produce syntactically relevant outputs; and (ii) our method is not solely relying on knowledge in GPT-4's training data. We view these as fundamental ingredients to the success of our method, regardless of the relative simplicity of the underlying benchmarks. Accordingly, our phase II evaluation, will focus on increasing the realism of the targets and employing state-of-the-art fuzzing techniques. **RQ3**, **RQ4**, and **RQ5** are less related to performance of our method as a fuzzer and are posed around

[3]While this question is very related to **RQ2**, the core difference is the focus on the diversity, rather than simply well-formedness. Additionally, in this research question, we do not assume the format has not been seen by the LLM.

**Table 1: Descriptions of the novel grammars we use to evaluate RQ2, in addition to the corresponding results. Here, for a character $c$ and string $w$, we use $\#c(w)$ to denote the number of occurrences of $c$ in $w$.**

| Description | Grammar | Accuracy |
|---|---|---|
| Modified Tomita-1 | ('zizzer'\|'zazzer'\|zuzz)* | 100% |
| Tomita-2 | (zq)* | 100% |
| Tomita-3 | All strings without an odd number of z's followed by an odd number of q's as a substring | 100% |
| Tomita-4 | All strings without *qqq* as a substring | 100% |
| Tomita-5 | $\#\text{'*'}(w), \#\text{'\$'}(w) \equiv 0 \bmod 2$ | 61% |
| Tomita-6 | $\#\text{'*'}(w) \equiv \#\text{'\$'}(w) \mod 3$ | 38% |
| Tomita-7 | 'zizzer'*'zazzer'*'zizzer'*'zazzer'* | 100% |
| Unusual Matching Parenthesis | $S \rightarrow \epsilon \mid (S\}S \mid (S{>}S \mid [S\}S \mid [S{>}S$ | 68% |
| Vowels with Length-field | $\{w : s \in (\text{'a'}\mid\text{'e'}\mid\text{'i'}\mid\text{'o'}\mid\text{'u'})^*, w = s \oplus \lvert s\rvert\}$ | 75% |

potentially unique opportunities or challenges presented by the use of LLMs. As such, we defer these questions for the full evaluation.

For **RQ1**, we present a simple experiment fuzzing a restricted python URL parser as found in [53]. We use OpenAI's GPT-4 [39] with a temperature of 0.1 as the LLM, and OpenAI's text-embedding-ada-002 [3] as the embedding model. We segment the URL file specification by subsection, and use $k = 4$ in the greedy $k$-NN prompt augmentation strategy. The maximum recursion depth, $d_{\max}$ is set to one. Lastly, we seed the examples generated by our method into the mutation fuzzer from [53]. We then compare the branch coverage and coverage of our method to a random fuzzer, and the mutation fuzzer from [53] seeded with a well-formed URL. The average results of this experiment over 5 trials can be seen in figure 2. On URLS, our method outperforms both fuzzers in terms of coverage, and slightly out performs the mutation fuzzer on branch coverage. Additionally, our method reaches higher coverage faster than the mutation fuzzer and random fuzzer.

As GPT-4 was trained on massive amounts of web data, any useful e.g., real and widely deployed, format specification will likely[4] appear in its training data. As such to test **RQ2**, we hand-write synthetic "format specifications" i.e., formal languages described by a mix of symbols and natural language. The goal is to demonstrate that GPT-4 is independently capable of synthesizing examples that satisfy formal constraints from a description. To test GPT-4's synthesis ability, we prompt it with a short blurb describing a formal grammar, and ask it to generate 100 examples. We then check the validity of these examples, and report the accuracy. A description of each file format, its formal grammar, and GPT-4's accuracy synthesizing examples from it, can be seen in table 1. For the exact prompts see the appendix. The first seven languages are modified versions of the Tomita grammars [48] which are commonly used as benchmarks for regular language induction [51][27] *inter alia*. The unusual matching parenthesis language is a modified version of the well-known Dyck-language, with the modification that the matching brackets do not "match" in a perceptual sense e.g., '[' can match with '}' or '>'. Finally, the vowels with length-field, is a clear example of a natural language constraint paired with a commonly used construct. With the exception of the Tomita-6 grammar, GPT-4 consistently performs well at example synthesis and is able to generate correct strings the majority of the time. While imperfect scores in the $60\% - 75\%$ range seem weak from a grammar learning or comprehension point of view, it is worth noting that for fuzzing purposes such mistakes can actually be beneficial. For example, $\approx 11\%$ of the incorrect strings GPT-4 synthesized for the vowels with length-field grammar, were of the form $w = s \oplus |s| + 1$, $|s| + 1 \leq 9$. In other words, on these strings GPT-4 counted the length field as contributing to the length of the string, and was off by one as a result (since $|s| + 1 \leq 9$). This could be viewed as a direct consequence of ambiguity in the prompt, or simply as an off-by-one error. Regardless, such mistakes are likely to be useful, or at least not actively harmful in fuzzing.

[4]Since the training data is not publicly released, we cannot anything definitive about what is or is not included.

## 6 RESULTS

This is a Stage 1 submission and results are not included.

## 7 PROPOSED EXTENSIONS & EXPERIMENTS FOR PHASE II

We plan to extend this work by expanding benchmarks. First and foremost we intend to perform ablation studies, determining the influence of each component of our method. For example, evaluating the performance of the *form* step of our prompting method (**RQ4**) by using antlr [1] to generate a wide range of parsers such as `http`, `mailto`, `fol` (first order logic), `csv`, `icalendar`, `xml`, to determine how many of the generated strings are well-formed. In a similar manner, we will investigate how effective the *deform* step is in inducing targeted errors in the parsers (**RQ3**). Finally, we plan to investigate the effect of the LLM's stochasticity on our method in **RQ5**, by studying how sampling temperature of affects its performance on creating well-formed and malformed seeds.

Next, we plan to expand the breadth and realism of fuzz targets by investigating whether our method can find bugs in real code. For this, we will use Magma [22] – an open source fuzzing benchmark with seven targets made up of real programs with previously documented bugs that were manually introduced into the latest version of the target. Magma's tooling records useful benchmarks, such as the raw number of bugs discovered, along with the time to reach and to trigger a bug. The increased complexity of the targets will

require that we extend the length of our fuzzing campaigns well beyond the current limit, and will necessitate the use state-of-the-art, open source, mutation fuzzers such as AFL [52], FairFuzz [31], Hongfuzz [2], or LibFuzzer [15] in our method and benchmarks.

## 8 CONCLUSION

We have proposed a natural language processing (NLP) augmented approach to fuzzing parsers. Our method uses a large language model (LLM) to recursively examine a natural language format specification to generate and mutate strong seed examples for a mutation fuzzer. In our preliminary experiments, we showed that this method outperforms a simple mutation fuzzer and a random fuzzer. We also used novel, handwritten synthetic formats to show that GPT-4 is capable of synthesizing examples from instructions it has not seen before, providing support that the output of our method does not need to rely on information GPT-4 memorized.

## ACKNOWLEDGMENTS

This work was partially supported by DARPA Safedocs Program award HR001119C0075 for which SRI is the prime contractor and Dartmouth is a subcontractor.

## A APPENDIX

### A.1 LLM Prompts

**Question Prompt**

What is the structure of a well-formed `object`? Provide your answer in JSON format with the name of each part as the key, and the value an English description of what the key is.

**Generate Prompt**

Generate an example of `object`. Provide your answer in JSON format where the key is the word 'example' and the value is the example of the object.

**Generate Prompt with Examples**

Generate an example of `thing` that is not `prior_things`. Provide your answer in JSON format where the key is the word `thing` and the value is the example of the `object`. If `prior_things_string` are the only values for `thing`, set the value to None.

**Generate Prompt with Examples**

Generate an example of `thing` that is not `prior_things`. Provide your answer in JSON format where the key is the word `thing` and the value is the example of the `object`. If `prior_things_string` are the only values for `thing`, set the value to None.

**Summarize Errors**

What are the parsing errors `file_type` parser should implement? Return your answer in JSON format where each key is an error and each value is a description of the error

**Mutating Strings**

Rewrite the string `target_example` so that it triggers an `error` error. Return your answer in JSON format where the key is the word `error` and the value is the string that triggers the error specified above.

**Compose Strings**

Make an example of a well-formed `parent_type` from, `“”.join(["the title: value" for title, value in children])`. Return your answer in JSON format where the key is the word `parent_type` and the value is the well-formed example.

## REFERENCES

[1] [n. d.]. ANTLR. https://www.antlr.org. Accessed: 2023-05-13.
[2] [n. d.]. Honggfuzz. https://honggfuzz.dev. Accessed: 2023-05-13.
[3] 2022. New and improved embedding model. https://openai.com/blog/new-and-improved-embedding-model#RyanGreene. Accessed: 2023-05-12.
[4] Sameed Ali, Prashant Anantharaman, Zephyr Lucas, and Sean W. Smith. 2021. What We Have Here Is Failure to Validate: Summer of LangSec. *IEEE Security & Privacy* 19, 3 (2021), 17–23. https://doi.org/10.1109/MSEC.2021.3059167
[5] Branden Archer and Darkkey. [n. d.]. Radamsa: a general-purpose fuzzer. https://gitlab.com/akihe/radamsa. Accessed: 2023-05-13.
[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[7] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu

**Tomita-1**

A well-formed string in the language L consists of any number of repetitions of the words 'zizzer', 'zazzer', or 'zuzz'. Generate 100 different strings from the language L. Make sure you only use the three rules above.

**Tomita-2**

Strings in the language L can only use the letters 'z' and 'q', and can have any number of repetitions of the word 'zq'
Generate 100 different strings from the language L.

**Tomita-3**

Strings in the language L can only use the letters z and q. The only restriction is that no string can have an odd number of z's followed by an odd number of q's.
Generate 100 different strings from the language L.

**Tomita-4**

A well-formed string in the language L consists of any string without three consecutive q's Generate 100 different strings from the language L.

**Tomita-5**

A well-formed string in the language L has an even number of *'s and an even number of $'s
Generate 100 different strings from the language L.

**Tomita-6**

A well-formed string in the language L has the property that the number of *'s minus the number of $'s is a multiple of 3 Generate 100 different strings from the language L.

**Tomita-7**

A well-formed string in the language L consists of:

- Any number of occurrences of the word 'zizzer'
- Followed by any number of occurrences of the word 'zazzer'
- Followed by any number of occurrences of the word 'zizzer'
- Followed by any number of occurrences of the word 'zazzer'

Generate 100 different strings from the language L. Make sure you only use the rules above.

**Unusual Matching Parenthesis**

A string in the language L is generated by the following rules:

- S = ""
- S = bracket1 + S + bracket2
- S = S + S

where bracket1 can be '(' or '[' and bracket2 can be '}' or '>'.
Generate 100 different strings from the language L. Make sure you only use the three rules above.

Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2022. On the Opportunities and Risks of Foundation Models. arXiv:2108.07258 [cs.LG]

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *ArXiv* abs/2107.03374 (2021).

[10] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[11] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. arXiv:2010.03150 [cs.LG]

[12] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. 2019. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 48–58. https://doi.org/10.1109/ICST.2019.00015

[13] Michael Eddington. [n. d.]. Peach Fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/. Accessed: 2023-05-14.

Vowel with Length Field

A well-formed string in the language L consists of any string made up of the letters 'a', 'e', 'i', 'o', 'u' with the length of the string at the end.
Generate 100 different strings from the language L.
Make sure you only use the description above.


[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[15] LLVM Foundation. [n. d.]. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2023-05-14.

[16] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*. 474–484. https://doi.org/10.1109/ICSE.2009.5070546

[17] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. *SIGPLAN Not.* 43, 6 (jun 2008), 206–215. https://doi.org/10.1145/1379022.1375607

[18] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 50–59.

[19] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf

[20] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: An Automatic Random Fuzzer for Common File Formats. *SIGPLAN Not.* 51, 12 (sep 2016), 13–20. https://doi.org/10.1145/3241625.2976017

[21] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[22] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. https://doi.org/10.1145/3428334

[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[24] Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. 2019. The Curious Case of Neural Text Degeneration. *ArXiv* abs/1904.09751 (2019).

[25] Srini Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. *ArXiv* abs/1808.09588 (2018).

[26] Dave Jones. [n. d.]. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity. Accessed: 2023-05-14.

[27] Anurag Koul, Alan Fern, and Sam Greydanus. 2019. Learning Finite State Representations of Recurrent Policy Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=S1gOpsCctm

[28] Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. CLAM: Selective Clarification for Ambiguous Questions with Generative Language Models. arXiv:2212.07769 [cs.CL]

[29] Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard. 2013. From Natural Language Specifications to Program Input Parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Sofia, Bulgaria, 1294–1303. https://aclanthology.org/P13-1127

[30] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMOSA: Escaping coverage plateaus in test generation with pre-trained large language models, In International Conference on Software Engineering. *ser. ICSE*.

[31] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[32] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2022. What Makes Good In-Context Examples for GPT-3?. In *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*. Association for Computational Linguistics, Dublin, Ireland and Online, 100–114. https://doi.org/10.18653/v1/2022.deelio-1.10

[33] Bill Marczak, John Scott-Railton, Bahr Abdul Razzak, Noura Al-Jizawi, Siena Anstis, Kristin Berdan, and Ron Deibert. [n. d.]. FORCEDENTRY: NSO Group iMessage Zero-Click Exploit Captured in the Wild. https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/. Accessed: 2023-05-13.

[34] Dheeraj Mekala, Jason Wolfe, and Subhro Roy. 2022. ZEROTOP: Zero-Shot Task-Oriented Semantic Parsing using Large Language Models. arXiv:2212.10815 [cs.CL]

[35] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[36] MITRE. [n. d.]. CVE-2014-0160 Detail. https://nvd.nist.gov/vuln/detail/cve-2014-0160. Accessed: 2023-05-13.

[37] Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. 2017. Faster Fuzzing: Reinitialization with Deep Neural Models. arXiv:1711.02807 [cs.AI]

[38] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. 2014. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14)*. Association for Computing Machinery, New York, NY, USA, 296–305. https://doi.org/10.1145/2664243.2664254

[39] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[40] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *Proceedings - 43rd IEEE Symposium on Security and Privacy, SP 2022 (Proceedings - IEEE Symposium on Security and Privacy)*. Institute of Electrical and Electronics Engineers Inc., 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[41] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-Based Whitebox Fuzzing for Program Binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 543–553. https://doi.org/10.1145/2970276.2970316

[42] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *ArXiv* abs/1711.04596 (2017).

[43] Patrick Schramowski, Cigdem Turan-Schwiewager, Nico Andersen, Constantin Rothkopf, and Kristian Kersting. 2022. Large pre-trained language models contain human-like biases of what is right and wrong to do. *Nature Machine Intelligence* 4 (03 2022), 258–268. https://doi.org/10.1038/s42256-022-00458-8

[44] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Sekhar Jana. 2018. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. *2019 IEEE Symposium on Security and Privacy (SP)* (2018), 803–817.

[45] Richard Shin, Christopher Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. 2021. Constrained Language Models Yield Few-Shot Semantic Parsers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 7699–7715. https://doi.org/10.18653/v1/2021.emnlp-main.608

[46] Siguza. [n. d.]. Psychic Paper. https://blog.siguza.net/psychicpaper/. Accessed: 2023-05-15.

[47] Sean W. Smith. 2012. Security and Cognitive Bias: Exploring the Role of the Mind. *IEEE Security & Privacy* 10, 5 (2012), 75–78. https://doi.org/10.1109/MSP.2012.126

[48] M. Tomita. 1982. Dynamic Construction of Finite Automata from examples using Hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*. Ann Arbor, Michigan, 105–108.

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

[50] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2018. Superion: Grammar-Aware Greybox Fuzzing. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2018), 724–735.

[51] Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 5247–5256. https://proceedings.mlr.press/v80/weiss18a.html

[52] Michal Zalewski. [n. d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2023-05-13.

[53] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2023. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2023-01-07 14:37:57+01:00.


Received 2023-05-15; accepted 2023-06-12; revised 15 May 2023; revised 15 June 2023; accepted 27 July 2023