Learning Program Models from Generated Inputs

Tural Mammadov
CISPA Helmholtz Center for Information Security
Saarland University
Saarbrücken, Germany
tural.mammadov@cispa.de

***Abstract*—Recent advances in Machine Learning (ML) show that Neural Machine Translation (NMT) models can mock the program behavior when trained on input-output pairs. Such models can mock the functionality of existing programs and serve as quick-to-deploy reverse engineering tools. Still, the problem of automatically learning such predictive and reversible models from programs needs to be solved. This work introduces a generic approach for automated and reversible program behavior modeling. It achieves 94% of overall accuracy in the conversion of Markdown-to-HTML and HTML-to-Markdown markups.**

***Index Terms*—software testing, security testing, reverse engineering, deep learning**

I. Introduction

New applications of ML have shown that they can characterize the behavior of complex programs and reverse the computation results without the need to understand the internal logic [1]. Traditionally, model learning relied on big generic datasets that were hard to maintain and unsuitable for program behavior learning. Today we can pair learners with grammars and generic test generators to produce domain-specific training datasets containing sufficiently diverse inputs.

Program behavior models will have a variety of applications in software engineering, such as:

- behavior mocking [2] - replicate the functionality of individual functions, modules, or even complete programs.
- test generation [3] - predict input specifications or program failing conditions given a program coverage or a call-graph; predict UI transitions given UI metadata.
- behavior differentiation [4] - detect implementation deviation among multiple revisions of the same app or across different apps that implement the same protocol.

If we learn program models not from input-output pairs but from output-input pairs, we will learn the behavior of the inverse program. Thus behavior models can serve as low-cost reverse engineering tools.

In this paper, we present Modelizer (Fig. 1) – a model extraction framework that uses grammars to automatically produce and verify inputs, as well as to model and decompose outputs from programs. The first experiments with Markdown-to-HTML and HTML-to-Markdown conversions are promising and show high accuracy.

II. Approach

We start behavior learning from the input generation. Our models train on artificially generated data, which solves the training data quality and quantity problems simultaneously. However, artificial data synthesis requires the availability of the corresponding generators that presumes all data specifications to produce valid inputs. To avoid the implementation of program-specific generators, we use grammars as producers [5]. Given the specification, such producers can generate an infinite amount of syntactically valid data. Such specifications can be predefined, encoded manually, or mined automatically [6] from the program under test.

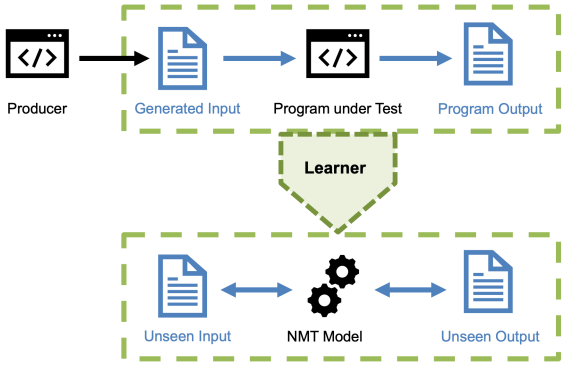

Fig. 1: The Modelizer Framework uses generated inputs to extract .input/output models from programs.

Then, we collect input-output pairs by processing the generated input with the program under test. Each data pair is later added to the training dataset, which is further passed to the Model Learner. Our Learner implements both Transformer [7] and Long Short-Term Memory (LSTM) [8] NMT architectures. After a predefined number of iterations, it produces the model, which can translate arbitrary input or output sequences. Soon, our models will learn and predict abstract *input features* like input grammar rules and constraints given a program *coverage*, a *call-graph*, or *side-channel* metadata.

III. Case Study

We have evaluated our approach by training a markup-conversion model of the Pandoc library [9] for the Markdown to HTML conversion task. Our study starts from the dataset generation task, where we manually define a grammar for Markdown strings. The Markdown Grammar is passed to Producer, which generates 10,000, 50,000, and 100,000 unique markdown structures. Each structure contains up to 12 top-level elements. Then the Pandoc library converts synthesized Markdown structures to HTML equivalents. Corresponding

Listing 1: Training Pairs. We train Modelizer with pairs of inputs and outputs as produced by the program

Markdown:	HTML:
### TEXT > TEXT > > > ***TEXT*** [*TEXT*](URL_LINK) `TEXT`	TEXT <blockquote> TEXT <blockquote> TEXT </blockquote> </blockquote> TEXT <code>TEXT</code>

Markdown-HTML structure pairs are added to the training dataset. Listing 1 contains an example of such structure pairs.

Since our Learner cannot process the whole document structure at once, we need to convert Markdown and HTML structures into sequences of smaller elements. The model's accuracy benefits from partitioning granularity. We have implemented two datatype-specific tokenizers that take each top-level element from a structure, split it into a sequence of tokens, which is further passed to the Learner. For each generated dataset, the Learner trains a model for five iterations.

We have also collected 10,000 previously unseen markdown-html document pairs for the model testing phase. The model precision was measured according to elementwise and document-wise prediction accuracy. If a document contains a single syntactically invalid top-level element, we mark the whole document as invalid. As you see in Table I, the best-performing model was trained on 100,000 markdown-html document pairs. While such NMT models show better performance by enlarging the training set size, in the current example, we demonstrate that such a *"lightweight"* model already achieves **94.03%** overall accuracy on the testing set. Listing 2 demonstrates an example of reverse HTML to Markdown translation.

Listing 2: Reverse Computation. The learned model can also map outputs to inputs with high accuracy

Input - HTML:	Output - Markdown:
Saturnushoc pascat medius excussaque Ceyx volucrem Dique in iam coniuge exige - Gravis languida Achilli labores - Curaque tua unda - Fessam diu summis furta corpore ramos 	# Saturnus hoc pascat medius excussaque Ceyx volucrem ## *Dique in iam coniuge exige* 1. Gravis languida Achilli labores 1. Curaque tua unda 1. Fessam diu summis furta corpore ramos

We still think that model accuracy will degrade when trained on more complex data types. However, we see two ways of improving model accuracy in such cases. First, our framework allows the infinite generation of more complex training samples on demand. Learners can benefit from it by continuously training until tolerable accuracy numbers are achieved. Alternatively, we can correct the invalid predictions by generating the smallest patches with the help of the Producer and evaluating the fixed prediction on a target program.

Table I: Model Accuracy

Precision	Training Set Size		
	10k	***50k***	***100k***
Elements	94.46%	97.64%	**98.29%**
Documents	80.62%	91.72%	**94.03%**

IV. Discussion

Once learned, such generic models are helpful in the software testing due to their reusability and simplicity of implementation. They can be used for the understanding of program or system behavior. The model querying does not depend on specific requirements and can be parallelized. Models can quickly adapt to the continuous changes of target programs by finetuning on newly generated samples. Such models are not intended to capture the behavior of complex systems at once. Instead, the chain of models can be used for emulating the data flow between subsystems.

Behavior modeling could serve as a replacement for program sandboxing. Program execution may be unsafe or require the availability of external resources and privileged access. In contrast, querying the program model does not depend on these additional factors. We can even learn such models when direct interaction with a program under test is impossible if enough input-output pairs were previously collected.

While we currently rely on manually-defined tokenizers, we look forward to automate this step by generating tokenizers from grammars or inferring tokenizers from counting vectorizers that incorporate neural semantic parsing. Also, existing natural language tokenizers still can be used with our Leaner.

V. Related Work

Katz et al. [1] first train a neural code decompiler from generated data. This approach is closest to ours. However, our framework allows behavior modeling of arbitrary programs, learns both direct and inverse models from programs, and generalizes the learning process from mapping input-output pairs to extracting common input properties and constraints.

"Learning to execute" [14] is one of the first attempts in neural behavior modeling, where a Python interpreter model was learned. The results were not promising because neural networks are inefficient in performing arithmetic operations. Similarly, the Transcoder model [15] suggests the usage of NMT models for translations of program source code from one programming language to another. Meanwhile, we support cross-protocol translations, like SVG images to Python code.

Several models, like CodeBERT [16] or CodeT5 [17], explore code understanding and generation problems. Such models are trained from code-to-text description pairs collected from publicly available datasets and repositories, which may contain malicious or faulty code snippets [18]. Instead, we are able to generate and test data on demand.

VI. Acknowledgment

Prof. Dr. Andreas Zeller from the CISPA Helmholtz Center for Information Security advises the author of this paper.

REFERENCES

[1] *O. Katz, Y. Olshaker, Y. Goldberg, E. Yahav,* "Towards Neural Decompilation" in arXiv:1905.08325, 2019.

[2] *Tim Mackinnon, Steve Freeman, and Philip Craig,* "Endo-testing: unit testing with mock objects" in Extreme programming examined. Addison-Wesley Longman Publishing Co., Inc., United States, May 2001, Pages 287–301.

[3] *Ari Takanen, Jared Demott, Charles Miller, Atte Kettunen,* "Fuzzing for Software Security Testing and Quality Assurance", Second Edition , Artech, 2018.

[4] *Michael Kart,* "Behavior-driven development: conference tutorial" in J. Comput. Sci. Coll. 27, 4, April 2012, page 75.

[5] *A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler,* "The Fuzzing Book", Retrieved 2023-01-07 14:37:57+01:00. https://www.fuzzingbook.org/html/GrammarFuzzer.html

[6] *R. Gopinath, B. Mathis, A. Zeller,* "Mining input grammars from dynamic control flow" in ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, November 2020, Pages 172–183.

[7] *A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin,* "Attention is all you need" in Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17). Curran Associates Inc., Red Hook, New York, United States, pages 6000–6010.

[8] *S. Hochreiter, J. Schmidhuber,* "Long Short-Term Memory", Neural Comput. 9(8), November 15, 1997, pages 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[9] *J. MacFarlane,* Pandoc - a Haskell library for converting from one markup format to another. Version 2.19. https://github.com/jgm/pandoc/

[10] Language Integrated Query (LINQ) (C#) by Microsoft. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/. Retrieved 2023-01-30 12:20:37+01:00

[11] ISO/IEC 9075-1:2016, Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework), International Organization for Standardization, Geneva, Switzerland, December 2016.

[12] Scalable Vector Graphics (SVG), Version 2.0, World Wide Web Consortium, Cambridge, Massachusetts, United States. April 10, 2018 https://www.w3.org/TR/SVG2/

[13] Mathematical Markup Language (MathML), Version 3.0, World Wide Web Consortium, Cambridge, Massachusetts, United States. April 10, 2014. https://www.w3.org/TR/MathML3/

[14] *W. Zaremba, I. Sutskever,* "Learning to Execute" in arXiv:1410.4615, 2014.

[15] *B. Roziere, M. Lachaux, L. Chanussot, G. Lample,* "Unsupervised translation of programming languages" in Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20), Curran Associates Inc., Red Hook, NY, USA, 2020 Article 1730, pages 20601–20611.

[16] *Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou,* CodeBERT: A Pre-Trained Model for Programming and Natural Languages in EMNLP 2020: Findings of the Association for Computational Linguistics, November 2020, Pages 1536-1547

[17] *Y. Wang, W. Wang, S. Joty, S. Hoi,* "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation" in EMNLP 2021: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, November 2021, Pages 8696-8708

[18] *R. Schuster, C. Song, E. Tromer, V. Shmatikov,* "You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion" in 30th USENIX Security Symposium. 2020, Pages 1559-1575.