DeepFuzzSL: Generating Simulink Models with Deep Learning to Find Bugs in the Simulink Toolchain

Sohil Lal Shrestha
Computer Science & Eng. Dept.
University of Texas at Arlington
Arlington, Texas, USA

Shafiul Azam Chowdhury
Computer Science & Eng. Dept.
University of Texas at Arlington
Arlington, Texas, USA

Christoph Csallner
Computer Science & Eng. Dept.
University of Texas at Arlington
Arlington, Texas, USA

ABSTRACT

Testing cyber-physical system (CPS) development tools such as MathWorks' Simulink is very important as they are widely used in design, simulation, and verification of CPS models. Existing randomized differential testing frameworks such as SLforge leverages semi-formal Simulink specifications to guide random model generation. This approach requires significant research and engineering investment along with the need to manually update the tool, whenever MathWorks updates model validity rules. To address the limitations, we propose to learn validity rules automatically by learning a language model using our framework DeepFuzzSL from a existing corpus of Simulink models. In our experiments DeepFuzzSL consistently generated over 90% valid Simulink models and also found 2 bugs in Simulink version R2017b and R2018b confirmed by MathWorks Support.

ACM Reference Format:
Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. 2020. DeepFuzzSL: Generating Simulink Models with Deep Learning to Find Bugs in the Simulink Toolchain. In *Proceedings of 2nd Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest '20)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cyber-physical systems (CPS) are integration of cyberspace and physical world through a network of interconnected components such as actuators and sensors. Engineers typically prototype CPS with graphical block diagram using commercial development tools such as MathWorks Simulink [29] (a de-facto industry standard), which enable them to model, simulate and analyze their system. Furthermore, these toolchain can automatically generate embedded code that are often deployed in target hardware of safety critical systems. It is thus very important to find and remove bugs in such development toolchains.

In software engineering, there are a number of ways to find bugs. Ideally one can formally verify the entire Simulink toolchain, but it is not feasible due to its large and complex code base and lack of complete formal specification, which can be partly attributed to its commercial nature [11]. Like many other software systems, toolchain testing suffers from the test oracle problem [2].

An alternative is fuzzing, or random test case generation which is an effective way to identify bugs [6, 7]. State-of-the-art Simulink-testing tool *SLforge* combined randomized fuzzing with differential testing and found 8 new bugs in Simulink [11]. Since Simulink does not have complete publicly available language specification, Chowdhury et al. [11] parsed semi-formal specifications from Simulink's web page automatically and rigorously incorporated them in SLforge's random model generator. While SLforge is proven effective, it inherently relies on documented specification to update it's random model generator.

To overcome the engineering effort of maintaining the tool with respect to subtle specification changes and adding new features while also preserving reasonable fidelity to the real world Simulink models, we propose to build a neural network model that can automatically generate Simulink models by learning directly from third-party Simulink models. We hypothesize that a neural network model should be able to capture undocumented Simulink specifications that is missed by earlier approach. The hypothesis is motivated by recent development in deep learning and natural language processing research that have constructed probabilistic language models of how humans write code. Such approach have shown efficacy of random program generation without the need of rigorously defining rules or grammar in a random program generator [15, 25]. For e.g., *DeepSmith* [15], a deep learning based fuzzer, have reported 50+ bugs in OpenCL compiler such as LLVM and claimed that it can be easily extensible to other programming languages with minimum engineering efforts.

Earlier work on applying deep learning to compiler fuzzing have mostly focused on programming languages (such as C, OpenCL) whose complete specifications are publicly available. In contrast, we focus on Simulink that lacks complete specification making it a better candidate to validate language agnostic deep learning framework that earlier work claims [15].

In this work, we portray random Simulink model generation task as a language modeling problem (Section 2.1). Traditional statistical language model approach like n-grams fails to capture semantic relations, thus is not useful in our work. In contrast, neural language model (Section 2.1) captures the semantic and syntactic structure of a given language. While there are different types of neural network architecture (such as feed forward, convolutional, recurrent etc), we chose Long Short Term Memory(LSTM) [18], a variant of recurrent neural network, which has proven effective in language modeling [32].

In our DeepFuzzSL framework, we extend DeepSmith architecture to generate random Simulink models. In doing so, we verify their earlier claim and validate our hypothesis. In our preliminary evaluation, our trained DeepFuzzSL model is able to generate over 90% valid Simulink models and have found 2 bugs in Simulink versions R2017b and R2018b confirmed by MathWorks Support.

To summarize, this paper makes the following major contributions.

DeepTest '20, May 25, 2020, Seoul, Republic of Korea
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- To best of our knowledge, this is the first work that employs LSTM to automatically generate Simulink models to test the Simulink toolchain.
- In our experiment, DeepFuzzSL found 2 confirmed bugs in the widely used CPS development tool Simulink, one of which is missed by previous state-of-the-art.
- Our DeepFuzzSL prototype implementation and evaluation data are open source at GitHub [37].

2 BACKGROUND

This section provides necessary information on neural language model, CPS models and commercial CPS tool chain Simulink.

2.1 Neural Language Model

Language modeling is the task of predicting the next word in a sequence based on the words already observed in the sequence. In essence, a language model assigns probability to a sequence of words, which is expressed as a joint probability over the words as:

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \prod_{i=2}^{n} P(w_i | w_{i-1}, w_{i-2}, \ldots, w_1),$$

where w_i is the i-th word in a sentence of length n. So given a arbitrary word sequence $(x_1, x_2, \ldots, x_t)$, a language model can compute the probability distribution of the next word x_{t+1} as $P(x_{t+1}|x_t \ldots x_1)$, where x_{t+1} can be any word in a vocabulary V = $\{w_1, \ldots, w_{|V|}\}$.

Conventional language models such as n-grams look at fixed *consecutive* context window (or finite window of *consecutive* previous words) to predict the next word. These kinds of language model couldn't be conditioned over large context window without running into *out of memory* issue [33].

On the other hand, a neural language model uses a neural network architecture to learn a language model as a distributed representation of words [4, 39]. Further improvement on neural language model gave rise to recurrent neural networks that can retain a state that can represent information from an arbitrarily long context window achieving state-of-the-art result in language modeling as well as other sequential learning task [22].

Using a language model, one can generate sequence of words conditioned on previous words. This is relevant to textual programming languages such as C, where, for example, a variable use never comes before variable definition. Although Simulink models are designed using graphical block diagrams, textual representation of the model follow the norm, where a block information never comes before the connection (or line) information making language model a good fit for this work.

2.2 CPS Model and Simulink

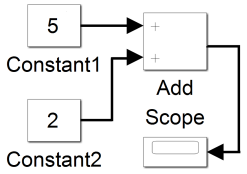

Figure 1: Example minimal toy Simulink model adding two constants, encoded in Listing 1.

```
Model { ...
Block {
  BlockType  Constant
  Name  "Constant1"... }
...
Line {
  SrcBlock "Constant1"
  DstBlock "Add" ... }
 ... }
```

Listing 1: Figure 1 model as text file (excerpt).

A CPS model is typically designed as a set of graphical *models* as seen in Figure 1. A model contains *blocks* that accepts data through the *input* ports. Block performs some operation on the data and can pass output to other blocks through *output* ports, using *connection* lines.

Simulink is a powerful, flexible, and de-facto standard commercial toolchain for CPS that supports various programming paradigms including data-flow and object oriented programming [40]. To design a CPS model, Simulink has support for various built-in block *libraries* [28]. Users can also create *custom-blocks* whose functionality can be defined through custom "native" code. Users can then *compile* and *simulate* a model. After compilation, Simulink offers different simulation modes [26]. In depth descriptions of CPS and Simulink can be found elsewhere [11, 30].

When a user attempts to open a Simulink model in Simulink, first the Simulink parser checks the model, possibly rejecting it and preventing the model from opening in Simulink. Once the model is opened, the user can compile and then simulate the model, which triggers different simulation phases [27]. In this paper, we consider a Simulink model to be *valid* if Simulink can open and compile the model without errors.

3 OVERVIEW AND DESIGN

DeepFuzzSL needs as input a set of seed Simulink models ("corpus"). Based on these seed models, DeepFuzzSL proceeds in five main processing phases, as shown in Figure 2, to encode the seed models and use the encoded seeds to train a generative ML model, sample from the trained ML model, and decode the samples back to Simulink.

3.1 Seed Models: Simulink Model Corpus

We identified two main options for representing Simulink models in files, *mdl* and *slx*. While the main format since 2012 has been *slx*, this format has several drawbacks for our purposes. Specifically, *slx* stores a given model as a sequence of XML files, which is verbose and requires reasoning about a set of files.

In contrast, the earlier *mdl* format is also text based but more compact and thus easier to parse and generate for a deep learner. Each model is contained in a single *mdl* file and there is tool support for conversion between *mdl* and *slx*. DeepFuzzSL thus uses *mdl*.

While more compact than *slx*, *mdl* is still much too verbose for state-of-the-art deep learning systems. For example, the *mdl* representation of the Figure 1 toy example consists of over 1 kLOC

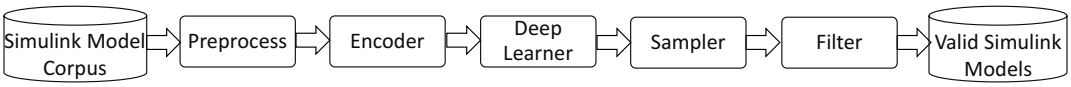

Figure 2: Overview of DeepFuzzSL's main processing phases.

with over 1,000 keywords and parameters following the structure shown in Listing 2. Before using a model as a seed for deep learning, we thus transform it as follows.

We remove *BlockDefaults {...}* and *AnnotationDefaults {...}* as such *mdl* file can be compiled without any issue. We also observed information of model component desgined by user in Simulink are stored inside *System*{...}. Thus we stripped down all other model parameter such as configuration defaults, graphical interface defaults ensuring that the model can be compiled (aka valid).

```
Model {
  <Model Param_Name> <Model Param_val>
  ...
  BlockDefaults {
    <Block Param_Name> <Block Param_val>
    ...
  }
  AnnotationDefaults {
    <Annotation Param_Name> <Annotation Param_val>
    ...
  }
  System {
    <System Param_Name> <System  Param_val>
    ...
    Block {
      <Block Param_Name> <Block Param_val>
      ...
    }
    Line {
      <Line Param_Name> <Line Param_val>
      ...
      Branch {
        <Branch Param_Name> <Branch Param_val>
        ...
      }
    }
    Annotation {
      <Annotation Param_Name> <Annotation Param_val>
      ...
    }
  }
}
```

Listing 2: Typical *mdl* file format representation

3.2 DeepFuzzSL Processing Phases

Following are DeepFuzzSL's five main processing phases.

Pre-processing: Before feeding the seed corpus of Simulink models to the neural network for training, we perform pre-processing steps so that we don't overburden the neural network to learn unnecessary rules that do not contribute to generate a valid Simulink model. We carry basic pre-processing steps such as white space removal, converting long block name to shorter ones, removing any annotations and location information as they are not required for its validity.

The *mdl* representation lists all "Blocks{ ... }" first and then the connection between them later in the file. During our initial experiments, our trained deep neural net model was only able to generate Simulink models containing just blocks without any connection between them. To mitigate this issue, we interleave block and connection (or line) information in the *mdl* file such that every pair of connected blocks are defined first followed by their connection information.

Encoder: Neural network requires numeric sequences as inputs. Hence all seed models are converted to a sequence of fixed size feature vectors, where each integer is an index of predetermined vocabulary. We also studied different ways source code is encoded. In [25], character level encoding of source code is adopted. This minimizes the vocabulary size but leads to very long sequences. On the contrary, token level encoding leads to shorter sequences but increases vocabulary size as every literal is uniquely represented. To demonstrate the two extremes, we ran an experiment with 30 unmodified Simulink models' *mdl* files, each consisting of 5 to 15 blocks and then extracted the number of tokens and vocabulary size in Table 1, with and without removing duplicate white space. To limit the size of the resulting feature vector, we encode Simulink models using a hybrid scheme that maps a few common keywords and parameters to tokens and the rest to characters.

Preprocess	Encoding	Tokens	Vocabulary
n/a	character	1,056,673	87
DWR	character	846,075	87
n/a	token	466,000	1,063
DWR	token	168,000	1,063

Table 1: Token count and vocabulary size of 30 Simulink models based on character and token level encoding; DWR = duplicate whitespace removal.

Deep Learner: We use a Long Short Term Memory (LSTM) network, a variant of recurrent neural networks following the success of many recent works [15, 24, 34]. We use a two layer LSTM network with 512 nodes per layer, which strikes a balance between the size of the neural net and the closeness of the learned distribution to the true distribution. This, in turn, yields a practical training time of the neural network. We defined the model using Keras [9] and Tensorflow [1] and open sourced the project on Github[1] for other researchers to train on their own corpus.

Sampler: After the training is complete, we seed the trained neural net with "Model {" tokens since every *mdl* file starts with it. Then we sample token-by-token to generate Simulink models. We halt the sampling when the opening and closing bracket counts become balanced or it reaches the maximum number of allowed tokens. Finally we decode the generated sequence back to text, which represents a Simulink model. Since we want to maximize the number of variations of generated models, we chose the next token from a randomized learned distribution, by performing a multinominal experiment.

[1] https://github.com/50417/DeepFuzzSL/releases

Filter: Lastly we filter out the generated Simulink models by opening and compiling them in Simulink. The valid Simulink models can then be used to test Simulink for crashes or be used for differential testing.

4 PRELIMINARY EVALUATION

Deep learning requires a large number of seed models. While there is existing work on a public corpus of third-party open-source Simulink models [13], these third-party models are quite diverse. It would have taken us significant work to normalize these existing models, to bring them into a unified shape useful for our deep learning setup. To side-step these issues, we instead trained our LSTM network on 1,000 SLforge-generated models.

We performed the training remotely in the high performance Texas Advanced Computing Center (TACC) [38]. Specifically, we used TACC's Maverick 2 cluster, which has support for GPU accelerated deep learning research workloads. We ran our experiments on a single Maverick 2 GTX node[2], which has 128 GB RAM, two 8-core 2.1 Ghz Intel Xeon processors and 4 NVidia 1080-TI GPUs.

Using the Adam optimizer [20], we trained the network for 400 epochs using gradient descent with a learning rate of 0.002, decaying 5% every epoch with mini-batch size 64. We selected these hyper-parameters (epochs, learning rate, decay, batch size) based on the best result after multiple experiment runs. On TACC's Maverick2 GTX nodes, training the neural network took some 2 hours.

As a preliminary evaluation, at this stage we focus on if it is possible to build on LSTM-based deep learning an effective approach for finding bugs in the Simulink toolchain. Specifically, we explore the following two research questions.

RQ1 Can a LSTM-based deep learning approach generate valid Simulink models?

RQ2 Can a LSTM-based deep learning approach find bugs in the Simulink toolchain?

4.1 Generating Valid Simulink Models (RQ1)

To evaluate our approach, we sampled 1,024 Simulink models from our trained LSTM network, limiting the maximum number of tokens in each generated sample to 5,000 (since the largest seed model also had 5k tokens) and reported the ratio of valid Simulink models (i.e., models Simulink compiles without warning).

To encourage variation in the generated sample Simulink models, we adapted the following three sampling strategies [19]. These strategies either re-scale the probability or restrict the set of tokens to be sampled from.

1. Sampling with Randomization ("Temperature Sampling"): Temperature sampling allows to control the variability of the next generated token while preserving the fidelity of the corpus to the learned distribution. In temperature sampling, we increase or decrease the probability of the most likely next token before sampling it. Basically the probability of the next token is controlled by a hyper-parameter called temperature (T) as:

$$P(x^{t+1}|x^t \dots x^1) = \frac{P(x^{t+1}|x^t \dots x^1)^{1/T}}{\sum_{x=V} P(x^{t+1}|x^t \dots x^1)^{1/T}}$$

A low temperature (less randomization) makes the language model increasingly confident in its top choices while infinite temperature (full randomization) corresponds to uniform sampling.

2. Top-k Sampling: In top-k sampling we order the tokens by probability and select the top k tokens. With $p' = \sum_{x=V_k} P(x^{t+1}|x^t \dots x^1)$, the original distribution is re-scaled as

$$P(x^{t+1}|x^t \dots x^1) = \begin{cases} P(x^{t+1}|x^t \dots x^1)/p' & \text{if x} \in V_k \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

3. Top-p or Nucleus Sampling: Similar to top-k sampling, we select the highest probability tokens whose cumulative probability mass is greater than p. Specifically, the top-p tokens form the smallest set such that

$$\sum_{x=V_p} P(x^{t+1}|x^t \dots x^1) >= p$$

The probability distribution is re-scaled similar to Equation 1.

Sampling type	Valid model %
Sampling with randomization(T), T = 0.8	92.8
Top-k, k = 10	92.5
Top-p, p = 0.9	94.4

Table 2: Ratio of valid Simulink models generated via various sampling strategies. The randomization, k, and p values were chosen based on experiments with best results; p = cumulative probability.

In all three sampling strategies, we sample the next token based on a multinomial experiment with the given probability distribution. In our experiment, the sampling time for each sampling strategy took around 13 minutes.

Table 2 summarizes our results. Overall, in all cases we observed over 90% valid generated Simulink models. In other words, Simulink could compile over 90% of the DeepFuzzSL-generated models without warning. Nucleus sampling performed better than the other two, which aligns with earlier results [19].

4.2 DeepFuzzSL Found Bugs in Simulink (RQ2)

We encountered six Simulink crashes (triggered by six DeepFuzzSL-generated models), of which five crashes occurred while Simulink opened a model and one crash occurred while Simulink compiled a model (after successfully opening it). So far we have reported the latter issue plus one representative of the five "crash while opening" cases to MathWorks via its bug report website[3].

For each reported issue we received email from a MathWorks Support person who investigated the issue and tried to the find the crash's root cause. Unlike open source projects, MathWorks does not list all issue reports or even all confirmed bugs on its website. The bugs listed on their web site do not show their corresponding Technical Support case (TSC) number.

Table 3 summarizes the two issues we have reported. *MathWorks Support* has confirmed both issues as bugs. Following are the details of these two bugs.

[2]https://portal.tacc.utexas.edu/user-guides/maverick2

[3]https://www.mathworks.com/support/bugreports/

TSC	Summary	Kind	MW
03322011	Simulink's parser fails to reject ill-formed model and crashes	O	K
03632450	Simulink's parser fails to reject model with ill-configured signal generator block	S	N

Table 3: Summary of issue reports; *TSC* = *Technical Support Case* number from MathWorks; *O* = issue when opening model; *S* = issue when simulating model; *MW* = feedback from MathWorks; *K* = known bug; *N* = likely new bug.

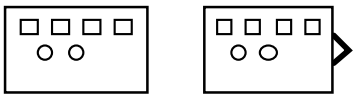

Figure 3: Signal Generator Block (a) DeepFuzzSL generated (b) from Simulink library

TSC 03322011: Invalid Input Model (Known Bug). This DeepFuzzSL-generated model consists of 3 discrete transfer function blocks. When trying to open this model Simulink crashes. Upon investigation, MathWorks Support determined that the generated model misses a certain parameter (OutputPortMap). MathWorks Support confirmed that this is a known bug. Instead of crashing, Simulink was supposed to produce an error and terminate normally.

TSC 03632450: Valid Input Model (Likely New Bug). This model generated by DeepFuzzSL consists of 25 blocks and 12 connections between them. Simulink could open this model normally without warning or errors. Simulink crashed when we tried to compile or simulate this model. In other words, DeepFuzzSL can generate models that pass Simulink's frontend parser.

Upon investigation, MathWorks Support provided as a reason that a signal generator block had a missing output port, which caused the crash. Figure 3(a) shows the signal generator block generated by DeepFuzzSL as opposed to one in Simulink library in Figure 3(b). MathWorks Support confirmed that this is a likely new bug. Instead of crashing Simulink should either produce an error or autofix the model. While the bug itself may be of low severity, it is an interesting one that validated our hypothesis "DeepFuzzSL can find bugs missed by SLforge". SLforge build a random model using Simulink block library, thus can not build Simulink models with such Signal Generator block.

5 RELATED WORK

Fuzzing is a well established testing and validation approach. Many test case generators use programming language's grammar to systematically generate syntactically valid programs. Textual programming language such as Java, C, Rust have random program generator that aimed at finding bugs in their respective compilers [3, 14, 16, 31].

While earlier work have largely focused on generating textual program (such as C , Java), limited work have been done for CPS models. CyFuzz [10] is perhaps the first tool to systematically generate Simulink models. As discussed throughout the paper, SLforge is the most closely related to our work. SLforge builds upon CyFuzz's limitations by incorporating informal Simulink specification into their random model generation. A subsequent work SLEMI [12] uses SLforge generated models to generate mutant of the seed model and found 9 confirmed bugs in Simulink. All of these work are tightly coupled with a particular CPS modeling language covering a subset of language specification and incurs high porting cost to other modeling language. In contrast, our work is loosely coupled with Simulink and has potential to cover undocumented specification.

Researchers are increasingly applying deep learning for software testing. Learn&Fuzz [17] learned from a corpus of PDF files to fuzz Microsoft Edge renderer. Closely related work DeepSmith [15] and DeepFuzz [25] learned probabilistic language model from a corpus of OpenCL and C programs and found multiple bugs in respective compilers. Both of the work target languages which have complete specification. On the contrary, although our work is built upon DeepSmith framework, we target CPS modeling language that does not have complete specification.

Similarly, while this paper looks for bugs in CPS tools such as Simulink using deep learning, a complementary line of work fuzz CPS models using machine learning and deep learning [5, 8, 21, 23, 35]. Liu et al. [23] use decision tree algorithm to stop test suite generation for fault localization of Simulink models. Chen et al. [8] use LSTM and Support Vector Regression to systematically guide generation of test suites for CPS network attacks. Their smart fuzzing system fuzzes actuator to drive CPS into unsafe state to diagnose cyber attacks. Kravchik et al. [21] study the use of convolutional and recurrent neural networks for detecting cyber-attacks in industrial control systems.

A summary of this work will also appear as a 2-page abstract in ICSE 2020's ACM Student Research Competition (SRC) [36]. In addition to the SRC summary, this paper adds details on sampling from the trained DeepFuzzSL model in Section 4.1 along with results in Table 2. Furthermore, this paper adds a description of the need for a hybrid encoding scheme with the results shown in Table 1. This paper also adds a *mdl* file structure and leverages the information while pre-processing that aid in training DeepFuzzSL in Section 3.1. This paper also includes details of the bug summary along with how we reported issues with MathWorks Support in Section 4.2.

6 CONCLUSIONS AND FUTURE WORK

Testing cyber-physical system (CPS) development tools such as MathWorks' Simulink is very important as they are widely used in design, simulation, and verification of CPS models. Existing randomized differential testing frameworks such as SLforge leveraged semi-formal Simulink specifications to guide random model generation which required significant research and engineering investment along with the need to manually update the tool, whenever MathWorks updates model validity rules.

To address the limitations, we proposed to learn validity rules automatically by learning a language model. Our framework DeepFuzzSL learned from existing corpus of Simulink models and generated valid Simulink models. In our experiments DeepFuzzSL consistently

generated over 90% valid Simulink models and also found 2 bugs confirmed by MathWorks Support.

Future work includes gathering a large Simulink model collection from public repositories such as Github and MathWorks File Exchange[4] and training the generative model on such a corpus as well as verifying the pre-processing heuristics.

ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. Christoph Csallner has a potential research conflict of interest due to a financial interest with Microsoft and The Trade Desk. A management plan has been created to preserve objectivity in research in accordance with UTA policy. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1527398 and 1911017 and a gift from MathWorks.

REFERENCES

[1] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.
[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525.
[3] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2016. Synthesizing Program Input Grammars. *CoRR* abs/1608.01723 (2016). arXiv:1608.01723 http://arxiv.org/abs/1608.01723
[4] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2001. A Neural Probabilistic Language Model. In *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.). MIT Press, 932–938.
[5] Anatolij Bezemskij, George Loukas, Diane Gan, and Richard J. Anthony. 2017. Detecting Cyber-Physical Threats in an Autonomous Robotic Vehicle Using Bayesian Networks. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*.
[6] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 180–190.
[7] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 197–208.
[8] Yuqi Chen, Christopher M. Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-guided network fuzzing for testing cyber-physical system defences. In *Proc. 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 962–973.
[9] François Chollet et al. 2015. Keras. https://keras.io.
[10] Shafiul Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner. 2016. CyFuzz: A differential testing framework for cyber-physical systems development environments. In *Proc. 6th Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy)*. Springer, 46–60.
[11] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge. In *Proc. 40th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM.
[12] Shafiul Azam Chowdhury, Sohil L Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. In *Proc. 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. To appear.
[13] Shafiul Azam Chowdhury, Lina Sera Varghese, Soumik Mohian, Taylor T. Johnson, and Christoph Csallner. 2018. A curated corpus of Simulink models for model-based empirical studies. In *Proc. 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*. ACM, 45–48.
[14] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience* 34, 11 (Sept. 2004).
[15] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *ISSTA 2018*.
[16] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 482–493.
[17] 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, ASE 2017*.
[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[19] Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. 2019. The Curious Case of Neural Text Degeneration. *CoRR* (2019). arXiv:1904.09751 http://arxiv.org/abs/1904.09751
[20] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proc. 3rd International Conference on Learning Representations (ICLR)*.
[21] Moshe Kravchik and Asaf Shabtai. 2018. Detecting Cyber Attacks in Industrial Control Systems Using Convolutional Neural Networks. In *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and PrivaCy, CPS-SPC@CCS 2018, Toronto, ON, Canada, October 19, 2018*. 72–83.
[22] Zachary Chase Lipton. 2015. A Critical Review of Recurrent Neural Networks for Sequence Learning. *ArXiv* abs/1506.00019 (2015).
[23] Bing Liu, Lucia, Shiva Nejati, and Lionel C. Briand. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 359–370.
[24] Xuan Liu, Di Cao, and Kai Yu. 2018. Binarized LSTM Language Model. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics(ACL): Human Language Technologies, Volume 1 (Long Papers)*. ACL, 2113–2121.
[25] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *The Thirty-Third AAAI Conference on Artificial Intelligence*. 1044–1051.
[26] MathWorks Inc. [n.d.]. How Accelerator Model Works Documentation. https://www.mathworks.com/help/simulink/ug/how-the-acceleration-modes-work.htm. Accessed Jan 2020.
[27] MathWorks Inc. [n.d.]. Simulation Phases in Dynamic Systems. https://www.mathworks.com/help/simulink/ug/simulating-dynamic-systems.html. Accessed Jan 2020.
[28] MathWorks Inc. [n.d.]. Simulink Block Libraries Documentation. https://www.mathworks.com/help/simulink/block-libraries.html. Accessed Jan 2020.
[29] MathWorks Inc. 2019. MATLAB & Simulink. https://www.mathworks.com/products/simulink.html/. Accessed Jan 2020.
[30] MathWorks Inc. 2019. Simulink Documentation. https://www.mathworks.com/help/simulink/. Accessed Jan 2020.
[31] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf
[32] Ngoc-Quan Pham, Germán Kruszewski, and Gemma Boleda. 2016. Convolutional Neural Network Language Models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*. 1153–1162.
[33] Raul Puri, Robert Kirby, Nikolai Yakovenko, and Bryan Catanzaro. 2018. Large Scale Language Modeling: Converging on 40GB of Text in Four Hours. In *30th International Symposium on Computer Architecture and High Performance Computing*. 290–297.
[34] Alec Radford, Rafal Józefowicz, and Ilya Sutskever. 2017. Learning to Generate Reviews and Discovering Sentiment. (2017). arXiv:1704.01444 http://arxiv.org/abs/1704.01444
[35] Sunny Raj, Sumit Kumar Jha, Arvind Ramanathan, and Laura L. Pullum. 2017. Testing autonomous cyber-physical systems using fuzzing features from convolutional neural networks: work-in-progress. In *Proc. 13th ACM International Conference on Embedded Software (EMSOFT) Companion*. 1:1–1:2.
[36] Sohil L Shrestha. 2020. Automatic generation of Simulink models to find bugs in cyber-physical system tool chain using deep learning. In *Proc. 42nd ACM/IEEE International Conference on Software Engineering (ICSE), Student Research Competiton (SRC)*. To appear.
[37] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. 2020. *50417/DeepFuzzSL: DeepFuzzSL First Release.* https://doi.org/10.5281/zenodo.3712482
[38] TACC at The University of Texas at Austin. 2018. Texas Advanced Computing Center - Homepage. https://www.tacc.utexas.edu/. Accessed Jan 2020.
[39] Kai Chen Gregory S. Corrado Jeffrey Dean Tomas Mikolov, Ilya Sutskever. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013*. 3111–3119.
[40] Marilyn Wolf and Eric Feron. 2015. What don't we know about CPS architectures?. In *Proceedings of the 52nd Annual Design Automation Conference*. 80:1–80:4.

[4] https://www.mathworks.com/matlabcentral/fileexchange/