LEARNING TO INFER RUN-TIME INVARIANTS FROM SOURCE CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

Source code is meant to be executed, as well as read. Developers reason about its run-time properties by inferring "invariants", which constrain program behavior; but they rarely encode these explicitly, so machine-learning methods don't have much aligned data to learn from. We propose an approach that adapts cues within existing if-statements regarding *explicit* run-time expectations to generate aligned datasets of code and *implicit* invariants. We also propose a contrastive loss to inhibit generation of illogical invariants. Our model learns to infer a wide vocabulary of invariants for arbitrary code, which can be used to detect and repair real bugs. This is complementary to trace-based methods, such as Daikon. Our results confirm that neural models can learn run-time expectations directly from code.

1 INTRODUCTION

Software developers make many inferences while reading code, about its performance, correctness, and run-time behavior. The latter is often summarized by *program invariants* – generic assumptions about code behavior, e.g. that a list index never escapes its bounds. Automatically inferring these is hard: predicting run-time behavior is undecidable, thus sound analyses are limited to small programs such as simple loops (Sharma et al., 2013a; Padhi et al., 2016). Ernst et al. (2007) pioneered learning them from highly informative *execution trace data*, but it requires access to realistic program inputs. Developers also rarely "assert" invariants explicitly, so static learning data is limited.

Yet these obstacles may be largely artificial. For one, practical programs rarely take on an exponential range of values, and developers reliably infer run-time constraints from a local context, using their past experience and cues from the code itself. What's more, developers do encode many run-time constraints in their code, in the form of if-statements. These guard blocks of code that should only execute under a given condition, such as that a map contains an element, or a parameter is not null. Interestingly, programmers rarely guard all possible conditions (e.g. nullity of all parameters), often rather relying on other (calling or called) methods to ensure them.

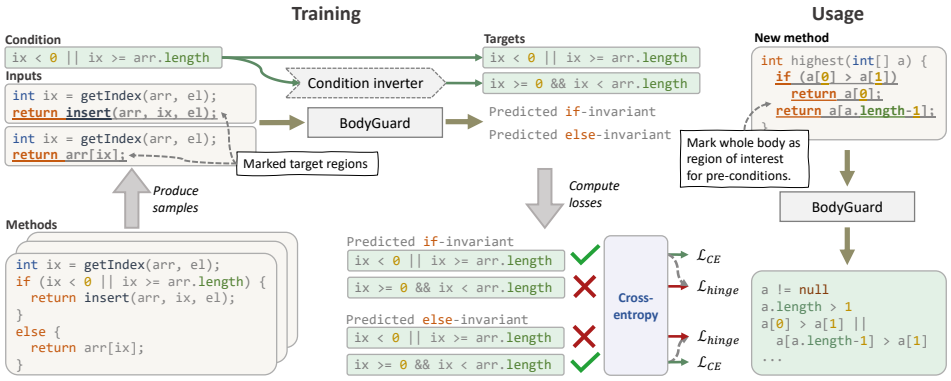


Figure 1: An overview of our learning approach. We extract samples from if-statements in Java methods by removing the guard and assigning it (or its negation, for the else block) as the target invariant(s) of the previously-guarded block. We train using the cross-entropy, plus a contrastive loss of this entropy relative to that of its logical negation (per sample).

Our central claim is that the natural distribution of programs includes many groups of similar functions, some of which assert run-time assumptions explicitly, and with much detail, while others vary along these dimensions. If true, we can use *explicit* conditional checks that guard blocks in functions to teach our models about the *implicit* invariants of unguarded blocks in similar functions. Furthermore, we conjecture that in such comparable samples, the condition is both *salient* (since it is checked explicitly) and *natural* (since it is written by humans).

Figure 1 shows how we exploit this symmetry between explicitly and implicitly guarded code: we remove explicit guards to generate the former. By removing these one at the time and filtering out implausible programs, we generate millions of realistic samples of functions and (now-)implicit run-time constraints. Our model, BODYGUARD, predicts a rich vocabulary of conditions about arbitrary code from new projects, and can be used to find & fix real missing-guard bugs with over 69% (repair) precision at 10% inspection cost. It also predicts more than two-thirds of Daikon's invariants that could previously only be inferred with run-time data, and some entirely new ones that can be validated automatically with trace data. This is a significant next step in learned static analysis, being the first to reliably produce natural invariants from arbitrary code alone. More generally, we show that learned models can implicitly represent behavioral semantics, just from code.

2 APPROACH

We train our model using a new loss function on a large corpus of open source data and evaluate it on two downstream tasks. For conciseness, we provide a high-level overview of our approach here.

2.1 DATASETS

We generate samples following Figure 1 from top-starred Java projects from Github, which we split by organization into training (920 projects), held-out (19 projects), and test data (61 projects). We extract all methods from each Java file and generate one sample for each (side-effect free) if-statement by removing the guard and storing its condition. As Figure 1 shows, this produces one or two equivalent code fragments for which the statement's condition (or its negation) is a pre-condition. The resultant sample contains the entire method (minus conditional check) as context, with the *range* of tokens where the invariant condition applies indicated.

Our model generates run-time conditions for an indicated range of code in a method. We evaluate it in two settings. First, we collect real missing if-guard bugs for our model to repair from 10K Java Github projects, by parsing their ($\sim$8M) commits for ones that a) introduced exactly one if-guard, and b) are described as bug-fixing (Ray et al., 2016). We find ca. 3K of these. Second, we use Daikon (Ernst et al., 2007) to collect execution trace data from a smaller set of eight projects that we manually instrumented, to compare our predictions to both Daikon's own and to traces directly. This helps us both assess the validity of our invariants on real executions, and, more generally, understand the *inference gap* between static and dynamic information for source code analysis.

2.2 MODEL SETUP

BODYGUARD uses a Transformer encoder-decoder model (Vaswani et al., 2017) to translate methods to invariants (see Figure 2). Syntactic & semantic code metadata, e.g. the AST and data-flow relations are known to help code understanding (Allamanis et al., 2018), thus we incorporate them using the relational attention mechanism from Hellendoorn et al. (2020). It adds a relation-specific bias term b^r_{ij} into the query-key comparison of scaled dot-product attention: $e_{ij} = (\mathbf{q_i} + b^r_{ij})\mathbf{k_j}^\top/\sqrt{N}$. The (learned) bias is sensitive to known relations r between tokens i and j (if any, and summed if multiple), to allow the model to selectively sharpen (or dampen) the significance of each relation. With 512-dimensional states, 8 heads, and 8 layers on both sides, our model has $\sim$67M parameters.

We created our own program graph extractor for Java, extracting 5 common semantic edge types, such as "last-use", "next-token", and "computed-from". We use the same "leaves-only" representation as Hellendoorn et al. (2020) to limit our inputs to tokens only, by rerouting edges that connect non-terminal nodes to their representative syntax token (e.g. from an if-statement node to its "if" code token). Finally, we use the relational mechanism between the decoder and encoder, to produce only invariants for the relevant range of tokens (using a unary relation, i.e., "is part of range").

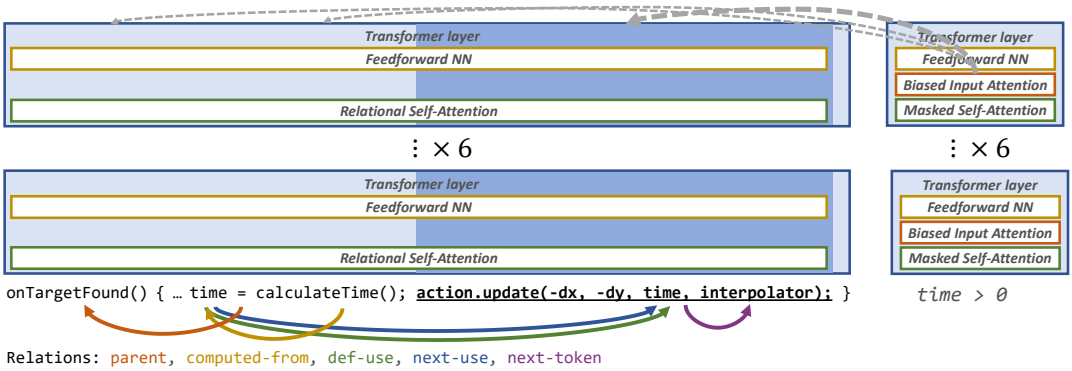
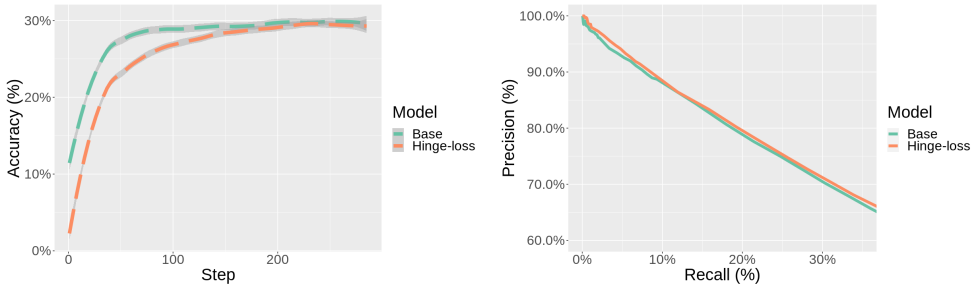


Figure 2: Schematic overview of our model: an 8-layer Transformer encoder/decoder, augmented with relational data (Hellendoorn et al., 2020), and a decoder bias towards the invariant's range.



(a) Accuracy on held-out data during training.

(b) Precision-Recall response of the trained models to limits on the entropy of generated invariants.

Figure 3: Model performance during and after training, focusing on the high-precision/low-recall domain for the second (overall test accuracies: 33.8% base, 34.7% hinge-loss).

2.3 Decoding Logical Statements

In conditions, small syntactic differences lead to drastic changes in run-time behavior. Removing esp. **`if`–`else`** statements as we do naturally yields code fragments with very similar, but logically opposite (e.g. '`!=` **`null`**' vs. '`==` **`null`**') conditions. We supervise our model to encourage its representations for *syntactically close* but *semantically opposite* statements to be distinct by introducing a *contrastive hinge loss* term. For every training sample, we also decode the invariant's logical negation, but require it to have a higher entropy than the real target. Concretely, given a statement inv with tokens t_i and a negating function neg, we use the cross-entropy loss $\mathcal{L}_{\text{CE}}$:

$$\mathcal{L}_{\text{CE}}(inv) = -\sum_{i=1}^{|inv|} \log \Pr(t_i \mid t_1 \cdots t_{i-1}, context) \tag{1}$$

to compute the entropy distance w.r.t. its negation:

$$\Delta_{inv} = \mathcal{L}_{\text{CE}}(\text{neg}(inv)) - \mathcal{L}_{\text{CE}}(inv) \qquad \mathcal{L}_{\text{hinge}}(inv) = \max\left(0, \Delta_{inv} - \epsilon\right)^2 \tag{2}$$

in which ϵ is the minimum desired entropy "distance" in bits. In this work, we set $\epsilon = 2$. For this *hinge-loss model*, as we will call it in the rest of this paper, we train with a loss equal to $\mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{hinge}}$.

3 Analysis

We sample our two models' held-out performance every 100,000 training steps, which produces the learning curves in Figure 3a. The hinge loss model saturates slower, as it faces the more challenging task of discriminating similar statements. However, after ca. one week of training, both models converge to approximately the same quality. It speaks to the challenge of this strategically important task that the models reach just ∼30% accuracy, due in part to the diverse vocabulary of invariants.

Figure 3b shows the evaluation on the test data, namely precision/recall behavior of the two models when ranking their predictions (from beam search) by entropy. Both models converge to (near) perfect precision at a commensurate expense of recall (breaking 80% precision around 20% recall). The hinge-loss model has a decidedly better precision/recall tradeoff, as well as higher overall accuracy.

3.1 MISSING IF DETECTION

We apply our models to predicting missing if-guards, first given a localized buggy segment (top row of Table 1). This most directly relates to our training signal, where we provided the location of the guarded code. Our model achieves 29.3% accuracy (base model: 26.8%), with a favorable 69.1% precision at 10% – enough to fix 215 out of 311 bugs once located. It is likely beneficial that this task is roughly as challenging as our test one: automatically synthesized training data is often overly easy compared to real tasks, which harms generalization (Hellendoorn et al., 2019b). The missing condition in these samples is (arguably) the most *salient* invariant in the entire method, not just the indicated code block, which our model should be able to prioritize. The next three rows of Table 1 show the results of running our invariant generator on every contiguous segment (up to 5 blocks) of code in each buggy method, ranking the top invariants across segments for inspection. This is substantially harder than the previous task, reducing the overall accuracy threefold and roughly halving precision. Nevertheless, that is still much better than might be expected if BODYGUARD had no location-sensitivity: we test over 30 blocks per method on average.

Table 1: Bug-detection and repair results on finding and predicting missing if-guards, across two settings: given the correct location, and across all possible locations, further analyzed by aspects of the top prediction.

Objective	**Accuracy**	**Top-5 Acc.**	**Precision @10% Recall**
Location given	29.3%	41.9%	69.1%
All Locations	10.4%	18.8%	39.9%
invariant correct	15.2%	24.2%	48.1%
position correct	19.4%	43.8%	100.0%

3.2 VALIDITY AND OVERLAP WITH DAIKON

Many of our emitted invariants are valid statements, even if not the real missing condition. We analyze this validity on methods for which we have trace data from Daikon (Ernst et al., 2007): our tool retrieves more than half of Daikon's invariants, and over two-thirds at 10% recall, from static code alone. In addition, many of our pre- and post-condition are not generated by Daikon (those out of its vocabulary, or with too few observations), even at a low entropy threshold. We assess these using a simple logical engine that compares various kinds of our invariants (e.g. collection inclusion, array length, nullity) against the recorded traces. We can check ca. 40% (12K) of our invariants, finding that our invariants are valid ca. 60% of the time at full recall, and this ratio increases greatly as we sharpen the entropy threshold, to over 80%, at recall values under 10%.

Many of our validated invariants were not produced by Daikon, implying that static and dynamic data are orthogonal for this task. We manually analyzed 708 pre-conditions that BODYGUARD generated at an entropy of $\leq$0.1; 540 of these could be checked with trace data, (449 valid, 91 invalid), and 122 of the remaining 168 were found valid on inspection. In effect, over 80% of our invariants at this recall level (3.5%) are correct, and more than two-thirds of the invalid remainder could be ruled out using trace data, if available, leaving a false positive rate of just 6.5% (46/708) when execution data is available (while also adding about 200 valid invariants to Daikon's predictions). This confirms that our tool is largely synergistic with dynamic, trace-based invariant generators.

4 CONCLUSION

Formal invariant inference with oracles or solvers (Sharma et al., 2013a;b; Padhi et al., 2016; Pham et al., 2017) does not scale beyond simple arithmetic programs. ML-based inference (Si et al., 2018; Hellendoorn et al., 2019a; Brockschmidt et al., 2017) scales better but still usually predicts invariant validity rather than generates them. In contrast, our work makes no assumptions about the code other than the availability of a parser, and is trained specifically to generate novel, likely invariants. It succeeds thanks to the observation that typically used invariants are in a sense *natural*, like many other aspects of programs (Hindle et al., 2012; Barr et al., 2013; Tsimpourlas et al., 2020), and therefore predictable from code alone, intentionally standardized for ease of reading and writing Casalnuovo et al. (2019). Previous approaches may be symbiotic with ours: trace data or SMT solvers can be used to filter invalid invariants, and possibly guide our invariant generation. Our approach can also support other code understanding tools that struggle to navigate an exponentially large program space.

REFERENCES

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 143–153, 2019.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=BJOFETxR-`.

Shay Artzi, Michael D Ernst, Adam Kiezun, Carlos Pacheco, and Jeff H Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. 2006.

Earl T Barr, Christian Bird, and Mark Marron. Collecting a heap of shapes. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 123–133, 2013.

Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 169–190, 2006.

Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. Learning shape analysis. In *International Static Analysis Symposium*, pp. 66–87. Springer, 2017.

Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. Do people prefer "natural" code? *arXiv preprint arXiv:1910.03704*, 2019.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773. ACM, 2017.

Vincent J Hellendoorn, Premkumar T Devanbu, Oleksandr Polozov, and Mark Marron. Are my invariants valid? a learning approach. *arXiv preprint arXiv:1903.06089*, 2019a.

Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 960–970. IEEE, 2019b.

Vincent J Hellendoorn, Petros Maniatis, Rishabh Singh, Charles Sutton, and David Bieber. Global relational models of source code. In *2020 8th International Conference on Learning Representations (ICLR)*, 2020.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847. IEEE, 2012.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: Open-vocabulary models for source code. *arXiv preprint arXiv:2003.07914*, 2020.

Miryung Kim and Andrew Petersen. An evaluation of daikon: A dynamic invariant detector.

Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In Andrew P. Black (ed.), *ECOOP 2005 - Object-Oriented Programming*, pp. 504–527, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31725-8.

Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 51(6):42–56, 2016.

Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 87–102. ACM, 2009.

Long H Pham, Ly Ly Tran Thi, and Jun Sun. Assertion generation through active learning. In *International Conference on Formal Engineering Methods*, pp. 174–191. Springer, 2017.

Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 93–104. ACM, 2009.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 428–439. IEEE, 2016.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. Verification as learning geometric concepts. In *International Static Analysis Symposium*, pp. 388–411. Springer, 2013a.

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *ACM SIGPLAN Notices*, volume 48, pp. 391–406. ACM, 2013b.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*, pp. 7762–7773, 2018.

Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. Understanding user understanding: determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 188–198. ACM, 2012.

Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. Learning to encode and classify test executions. *arXiv preprint arXiv:2001.02444*, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.