LEVERAGING STATIC ANALYSIS FOR BUG REPAIR

Ruba Mutasim Gabriel Synnaeve David Pichardie Baptiste Rozière
Meta AI
broz@meta.com

ABSTRACT

We propose a method combining machine learning with a static analysis tool (i.e. Infer) to automatically repair source code. Machine Learning methods perform well for producing idiomatic source code. However, their output is sometimes difficult to trust as language models can output incorrect code with high confidence. Static analysis tools are trustable, but also less flexible and produce non-idiomatic code. In this paper, we propose to fix resource leak bugs in IR space, and to use a sequence-to-sequence model to propose fix in source code space. We also study several decoding strategies, and use Infer to filter the output of the model. On a dataset of CodeNet submissions with potential resource leak bugs, our method is able to find a function with the same semantics that does not raise a warning with around 97% precision and 66% recall.

1 INTRODUCTION

Detecting and repairing bugs in source code necessitates strong reasoning skills over code structures. Current models struggle to capture such reasoning in the absence of a sufficiently large and reliable dataset of bug fixes. Several methods rely on mining GitHub to extract bugs and their corresponding fixes using some forms of heuristics (Tufano et al., 2019; Chen et al., 2019; Gupta et al., 2017), which can often introduce noise into the extracted dataset. Other works rely on synthetically created bugs (Hellendoorn et al., 2020; Allamanis et al., 2021; Pradel and Sen, 2018; Yasunaga and Liang, 2020). While they are able to generate a large amount of parallel data, their examples do not always match real bugs found in real code. There are also some fundamental challenges associated with using machine learning models for bug repair. These challenges stem from the fact that these models can make subtle or unexpected errors, and it is often difficult to trust the output of a machine learning model without manual checks. Static analysis tools, on the other hand, produce reliable results, but are frequently limited in scope and rarely capable of proposing idiomatic fixes to the source code.

Infer is an open-source static analysis tool developed at Meta and used in industrial settings. It targets mobile apps written in Java/Kotlin or Objective C, and also C++ code used in website backends. It already reported 100,000 issues (Distefano et al., 2019). At the moment, bugs revealed by Infer are fixed manually by developers. Tools proposing automatic code repair would help improve the bug-fixing rate. This often requested feature is difficult to add to static analysis tools like Infer, because they do not directly analyze source code, but rather a low level representation (similar to LLVM IR (Lattner and Adve, 2004)). In some situations, proposing a repair at IR level is feasible, but still useless from the programmer point of view, as there are no tools to decompile IR to source code.

In this paper, we propose using machine learning in conjunction with infer (static analysis tool) to fix bugs and repair code. This approach can be used to solve bugs in real code. It leverages neural machine translation to generate fixed source code, while deferring the reasoning about bug detection and correction to Infer, which is much more reliable in the absence of high-quality code repair data. More precisely, we train a model to decompile Infer IRs to code, and use it to retrieve repaired source code from repaired IRs.

Our main contributions are:

- We generate a parallel dataset of Java code and Infer IRs, and train a model to decompile Infer IRs.
- We propose a reliable method to automatically fix resource allocation bugs in the Infer IR.

- We combine the new auto-fix tool and the decompiler to repair resource allocation bugs and recover fixed java source code. We use our new method to fix more than 66% of the Infer warnings in our dataset. The proposed fixes still pass the unit tests for 96.9% of the submissions.

2 Method

This section describes our method for automatically fixing resource leak bugs flagged by Infer in Java. We describe our dataset, modeling choice, and how the fixes are done automatically in the IR. Our modelization choices for IR inversion are largely inspirated by Szafraniec et al. (2022), who train a model decompiling LLVM IRs for code translation.

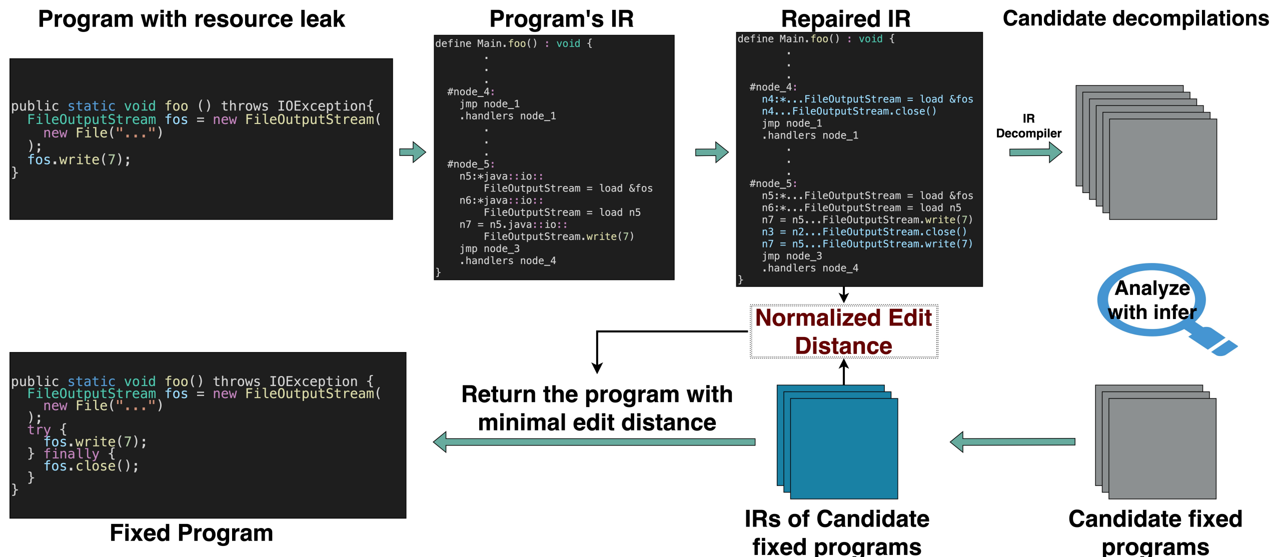

Figure 1: **Debugging using Infer IRs.** The code is compiled into the Infer IR, then fixed at the IR level and translated back to fixed source code using our model. We use the Infer and the java compiler to filter the generations sampled from the model. We use the normalized edit distance in IR space to select a single candidate to propose as a fix.

2.1 Data

GitHub BigQuery. For our experiments, we use two data sets. First, there is the publicly accessible GitHub dataset on Google BigQuery, which consists of a corpus of GitHub files. Similarly to Szafraniec et al. (2022), we only train on the portion of this dataset that can be compiled independently, so that we may create IRs to train the decompiler. This dataset was divided into three groups: training, validation, and testing.

CodeNet. CodeNet (Puri et al., 2021) is a large-scale AI for code dataset for learning a variety of coding tasks. It has about 14 million samples of code, each of which is meant to be a solution to one of 4000 coding problems. We only use the java portion of the dataset, and we similarly filter out non-compiling files. For this dataset, we divided it into training, validation, and testing based on the problem ids. The java files in the dataset were submitted to 3219 problems, and we randomly selected 300 ids for validation, 300 for testing, and the remaining 2619 for training.

We use every Java file from our datasets for pre-training. For tasks involving Infer IRs such as IR decompilation, we limited ourselves to files that compile in isolation for practical reasons. The number of files in our dataset is shown in Table 1.

Table 1: **Dataset.** This table shows the size of our dataset, in number of files, number of tokens and gigabytes. The full Java dataset is used for pre-training. The matched Java dataset corresponds to the subset of the full java dataset for which we were able to create an Infer IR. It is used for IR decompilation. The Infer IR dataset contains the matching IRs. It is used for both pre-training and IR decompilation.

	FULL JAVA	MATCHED JAVA	INFER IR
NB FILES	15.6 M	1.6 M	1.6 M
NB TOKENS	16.5 B	0.6 B	6.4 B
DISK USAGE	77.5 GB	2.3 GB	37.7 GB

Data preprocessing. We use Tree-sitter[1] to parse Java code into an AST and convert it to list of tokens. We tokenize Infer IRs by splitting them on white spaces and newlines. We used the same learned BPE (Sennrich et al., 2015) codes as TransCoder (Roziere et al., 2020) and DOBF (Lachaux et al., 2021)[2]. It is learned on AST-tokenized Java, C++ and Python code. We compile code using `javac`, get textual Infer representations using the `-dump-textual` options.

2.2 MODEL

Model architecture. We consider a sequence-to-sequence (seq2seq) transformer (Vaswani et al., 2017) model, which consists of an encoder and a decoder of 6 layers each and a total of about 312M parameters. We set the number of attention heads to 8, and the hidden dimension to 1024 (128 per head). We use layer dropout (Fan et al., 2019) with a probability parameter of 0.15 to make our training more efficient and reduce overfitting.

Model pretraining. Similarly to Szafraniec et al. (2022), we pre-train our model using the denoising auto-encoding (DAE) task (Vincent et al., 2008) similarly to Ahmad et al. (2021), pre-trained using the masked language modeling task similarly to Roziere et al. (2020). More precisely, we add noise to a sequence using a variety of procedures (e.g. shuffling, removal, masking subsequences) and train the model to retrieve the original sequence. We pre-train our model on our full Java dataset and on all the Infer IRs we were able to generate. DAE is an effective task to simultaneously train a model to understand IRs and generate Java source code.

IR inversion training. For the IR inversion task, we generate parallel Java/IR sequences with Infer and train the model to reverse this process. More precisely, we feed our model with the Infer IR and train it to retrieve each subtoken of the corresponding Java source code using the cross-entropy loss.

Code repair. In order to perform code repair, the model is used to decompile the automatically corrected IRs of flawed programs into a bug-free version of these programs. The corrected IRs are obtained automatically using the method described in Section 2.3. For this task, we favor improvements that require only minimal changes to the source code (buggy program). Therefore, for the decompilation of the repaired IRs, we select the best model using the edit distance.

Decoding strategies. We evaluate our IR inversion model using greedy decoding. For the bug repair models, we use either beam search decoding (Koehn, 2004) or nucleus sampling (Holtzman et al., 2019) with temperature 1.3, p-value 0.9, and test several number of samples to get candidate programs from the decompiler. Before proposing a fix, we filter the generations to keep only those that compile with `javac` and do not raise a warning with Infer. If several samples pass this filter, we compute their IRs again and select the one with the minimal edit distance compared to the fixed IR. If no sample compiles without any Infer warning, we do not propose a fix.

[1] `https://tree-sitter.github.io/tree-sitter/`

[2] `https://github.com/facebookresearch/CodeGen/tree/main/data/bpe/cpp-java-python`

2.3 AUTOMATIC IR FIXES

Infer reports about a broad collections of programming errors such as deadlocks, buffer overflows, dubious dead code, null or dangling pointer dereferences, execution time regressions, etc... Among them, **resource leaks detection** catches subtle bugs and is amenable to program repair. This analysis tracks Java objects, the **resources**, that are supposed to be closed when not used any more in a program. Failure to close them is a **resource leak** that unnecessarily consume limited system resources (like file or socket handlers) and may lead to system crashes in production. Correct closing of such resources requires to add enough call to a `close` method, taking into account exceptional execution paths with `try/catch/finally` blocks.

For example, the following program is missing a call to `fos.close()` in the case where there is an exception during `fos.write()`. It will abruptly end the execution of `foo` without executing the close statement.

```
public static void foo () throws IOException{
    FileOutputStream fos = new FileOutputStream(new File("..."));
    fos.write(7);
 }
```

To fix this problem, the programmer can use a `finally` block that is executed in both normal and exceptional paths, as follow :

```
 public static void foo () throws IOException{
    FileOutputStream fos = new FileOutputStream(new File("..."));
    try {
      fos.write(7);
    } finally {
      fos.close();
    }
}
```

At the IR level, resource leaks are detected at specific program points where all the following condition are met: i) the resource is still opened ; ii) the resource is still reachable (in the heap) via a local variable `res`; iii) on all paths from the current point to an exit (normal or exceptional) of the current function, the variable `res` is not mentioned any more. Such a program point is not only a right place to report for resource leak, but also to propose to add a close statement. Figure 4 in the Appendix presents an example of such a repair at IR level. The Java program at the top of the figure translates into the IR program on the left. Such a IR program is a list of instruction blocks. Each block has a label (`#node_0, ...`) and ends with jumps to other blocks (`jmp` instruction). The block `#node_1` is the exit block (with an empty list of successors). Some block also have a **handler** block (given with the keyword `.handlers`) which represents where the control flow will be rerouted if an exception is thrown. The program is missing two close statements and the IR on the right is the result of the automatic Infer fix. The first missing close is added after the normal execution of the `write` call, in block `#node_5`. The second close is added in the block `#node_4` that is the handler block of block `#node_5`. If the `write` call ends with an exception, the rest of block `#node_5` will not be executed and the control will jump to block `#node_4`. In such a situation, the method `foo()` will return an exception but the resource is properly closed before that.

2.4 EVALUATION

In this section, we define the metrics and methods used to evaluate our models for IR decompilation and bug repair. Except for the unit tests accuracy metric, we test the performance of our model for IR decompilation using the combined CodeNet and GitHub BigQuery datasets. The unit test accuracy and the metrics for bug repair are evaluated using only the part of the CodeNet for which we were able to extract working unit tests automatically.

IR inversion. We use the standard criteria for assessing machine translation models—perplexity, BLEU score (Papineni et al., 2002), and next token prediction accuracy—to evaluate the translation model's performance during training. Furthermore, we monitor a number of additional indicators to

buggy program	Proposed fix

`void reader() throws IOException {`
`    FileInputStream fis = new FileInputStream("file.txt");`
`    BufferedInputStream bis = new BufferedInputStream(fis);`
`    try {`
`        bis.read();`
`        }`
`    finally {`
`    }`
`}`
	`void reader() throws IOException {`
	`    FileInputStream fis = new FileInputStream("file.txt");`
	`    BufferedInputStream bis = new BufferedInputStream(fis);`
	`    try {`
	`        bis.read();`
	`        }`
	`    finally {`
	`        bis.close();`
	`    }`
	`}`

Figure 2: **Examples of function fixed by our pipeline.** The function on the left has potential resource leaks. We use infer to create the corrected IRs for this function, and then we use the decompiler to translate the corrected IR back to the original source code. The outcomes is an edited version of the source code that has no resource leak.

make sure the decompilation step introduces little to no modifications to the ground truth program's semantics. Concretely we evaluate the model on the following metrics, shown in Table 2:

- Mean normalized edit distance: the mean normalized edit distance (NED) between the ground truth program and the model's prediction. Normalized edit distance—as the name suggests—is the Levenshtein distance normalized by the length of the reference program.
- Compilation accuracy: the percentage of the model's output that compiles with Javac.
- IR exact match: percentage of retrieved source code that are compiles to exactly the same IR as the ground truth.
- Unit test accuracy: for the CodeNet dataset, we parse the unit-tests from the problem description and only consider samples in the dataset that pass all their unit tests. We then decompile the IRs of those samples and run the unit tests on the output. The unit test accuracy is the percentage of the outputs that pass all the unit-tests. This metrics tests whether the generations semantically match the input.

Bug Repair. We evaluate our model for bug repair on CodeNet submissions with working unit tests. We fix the resource leak automatically in IR space as described in Section 2.3. Then, we use our IR decompiler model to generate `N` candidate solutions from the fixed Infer IR, and run infer again to select successful fixes with no Infer alarm. Then, we select the top candidate among the successful fixes using the edit distance in IR space and run the unit tests. We evaluate our model for bug repair using these metrics:

- Fix proposed: percentage of times when our model was able to propose at least one fixed version with no Infer alarm
- Fix precision: percentage of times when the proposed fix passes all the unit tests in CodeNet
- Fix recall: percentage of times when our system proposed a correct fix, i.e. there was at least a solution raising no alarm and the selected fix passes the unit tests.

3 Results and Discussion

Table 2 displays the results of the decompiler on the combined CodeNet and GitHub BigQuery datasets. The version of our model pretrained with denoising auto-encoding reaches a slightly higher compilation rate than the model trained from scratch. It also reaches much higher scores on the IR match and unit test accuracy metrics, indicating that pretraining is especially beneficial for generating semantically valid decompiled code. Based on these metrics, we use the pretrained version of our decompiler for bug fixing.

Table 3 shows the performance of our model for fixing resource leak bugs from the CodeNet dataset for nucleus sampling and beam search with several sample sizes. As expected, increasing the number of samples increases the number of proposed fixes for both decoding strategies. With our temperature value of 1.3 (selected through grid search), nucleus sampling starts much lower than beam search but

its performance is essentially the same with sample size 100. The precision of the fix, measured using unit tests extracted from input and output examples from the CodeNet problem statement, tends to decrease with the number of samples. However, the recall still increases with the number of samples due to the increase in number of proposed fixes. We hypothesize that, with more samples, our model starts proposing fixes for harder code snippets, therefore decreasing precision but increasing recall.

Several methods could be studies to avoid the decrease in fix precision when increasing the number of samples. Our criterion to select the best candidate based on the edit distance between IRs may be lacking for large sample sizes. Additional methods, based on the number of the ratio of fix candidates that compile and raise no Infer warning could be used to decide whether to propose a fix. We could also develop better methods to evaluate the semantic equivalence of Infer IRs. Overall, the decoding strategy and number of samples should depend on requirements in terms of latency, computational costs, precision and recall. While beam search seems clearly preferable for low sample sizes, sampling is likely to outperform it for sample sizes above 100.

Our decompiler also proposes interesting improvements to the coding style. Consider the second example of Figure 3 in the Appendix: while the original code used the generic `Exception` class in the catch statement, the decompiled version of the example suggests using specific types of exceptions (i.e. `IOException` and `ClassNotFoundException`), which is generally regarded as better practice. Similarly in the first example of Table 3, the model extracts the value `4096` into a constant `BUFFER_SIZE = 4 * 1024`. In the third example, our model rewrites the modern try with resources[3] construct into a more classical `try` syntax. This type of refactors have no impact on the IR due to compilation-time optimizations.

Table 2: **IR inversion evaluations.**. This table shows our performance for decompiling Infer textual IR to Java source code. The token accuracy corresponds to the accuracy of the model when predicting the next token. NED stands for Normalized Edit Distance, or the Levenstein distance between the output and the ground truth divided by the length of the ground truth. These metrics were computed using greedy decoding. The unit test accuracy was computed on a subset of the CodeNet dataset containing only the functions with working unit tests. The performance of the model trained from scratch is close to that of the pretrained model for the perplexity, compilation rate, BLEU score and edit distance, but much lower for the IR match and unit test accuracy metrics. Even though an IR match implies that the semantics of the function are the same, the unit test accuracy score is lower than the IR match score for the model from scratch because the unit tests are run only on a subset of the dataset.

Model	From scratch	Pretrained
Perplexity	1.221	1.178
BLEU	79.4	78.9
Token Acc	96.1%	96.6%
Mean NED	0.034	0.035
Compil Rate	86.5%	89.7%
IR match	60.6%	71.2%
Unit test Acc	59.5%	86.9%

4 Related work

4.1 Machine learning for automatic bug repair

Based on mined bug fixes. Several studies use examples of manually fixed bugs to train machine learning algorithms. For instance, (Tufano et al., 2019) create a parallel dataset of buggy-fixed source code by selecting GitHub commits with words like "fix" or "solve" in their messages. Then, they train a model to translate from buggy to fixed code. SEQUENCER (Chen et al., 2019) is a similar model, which specialized in one-line repairs. It is trained on a mined dataset of (buggy program, line number of the bug, fixed line). DeepFix (Gupta et al., 2017) trains a neural network to address compilation errors, they use a compiler as an oracle to validate patch candidates before suggesting them to the user.

[3] `https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`

Table 3: **Debugging results for CodeNet submissions with working unit tests.** We show the performance of our model for bug fixing with both nucleus sampling and beam search and for several sample sizes. As expected, the number of proposed fixes increases with the sample size, as it increases the probability to find a Java function which is not flagged by Infer. The fix precision decreases slightly with the number of samples and the recall increases. Nucleus sampling is performed with temperature 1.3. These metrics are evaluated on 843 CodeNet submissions with working unit tests.

Decoding strategy	Fix proposed	Fix precision	Fix recall
Sampling 1	56.8%	97.3%	55.3%
Sampling 5	64.5%	97.4%	62.9%
Sampling 10	66.3%	97.3%	64.5%
Sampling 50	68.0%	97.0%	66.0%
Sampling 100	68.3%	96.7%	66.1%
Beam Search 1 (Greedy)	61.5%	97.9%	60.1%
Beam Search 5	65.6%	97.7%	64.1%
Beam Search 10	67.4%	97.2%	65.5%
Beam Search 50	68.6%	96.9%	66.4%
Beam Search 100	68.3%	96.7%	66.1%

DeepRepair (White et al., 2019) use auto-encoders to compute similarity scores between different code snippets to prioritize and transform repair components, and improve on redundancy-based program repair techniques. Another pattern-based method, called AVATAR (Liu et al., 2019), uses fix patterns of static analysis violations as components for patch generation. Getafix (Bader et al., 2019) proposed a new hierarchical clustering technique that arranges fix patterns into a hierarchy ranging from broad to narrow patterns. This clustering approach is used to quickly choose the best fix to recommend to the user.

Bhatia et al. (2018) combine neural networks and constraint-based reasoning to repair syntax errors. Dinella et al. (2020) cast the problem of program repair as learning a sequence of graph transformations. Their model makes a sequence of predictions including the position of buggy nodes in the program's graph and corresponding graph edits to produce a fix for Javascript programs. Vasic et al. (2019) develop a multi-headed pointer networks for locating and correcting variable misuse bugs. DeepDebug Drain et al. (2021) train a transformer model with back-translation, and fine-tune it using various program analysis information obtained from test suites.

Based on generated bugs. Other methods generate parallel buggy/fixed code pairs by introducing bugs automatically in correct code. For instance, BUGLAB (Allamanis et al., 2021) makes use of a selector model that decides which bug to apply to a code snippet and a detector model that detects and repair the inserted bug (if one was inserted). Yasunaga and Liang (2020) generate large amounts of bugs from correct code using several corruption rules. They train a graph-based neural network, dubbed Dr Repair on the obtained data. Ye et al. (2022) generate more than 1 million examples of training samples by perturbing code from open source projects. They outperform supervised repair approaches on Defects4J (Just et al., 2014).

Real-world datasets. There are a few manually curated bug fix datasets for Java (Just et al., 2014; Durieux and Monperrus, 2016; Saha et al., 2018; Madeiral et al., 2019; Bui et al., 2022) and other languages (Böhme et al., 2017; Tan et al., 2017; Hu et al., 2019). Due to their small sizes, they are generally used only for evaluation purposes.

Large language models. Large language models trained on code have recently demonstrated strong capabilities for detecting and resolving bugs in code, with PALM (Chowdhery et al., 2022) outperforming Dr Repair on DeepFix Gupta et al. (2017). They can also explain the bug and the proposed fix, which are useful features for programming assistants. (Prenner and Robbes, 2021; Kolak et al., 2022) study the capabilities of such models for code debugging on quantitative benchmarks.

4.2 Neural IR decompilation

Other works used neural networks for decompilation. Katz et al. (2019) train a sequence-to-sequence model to retrieve C code from either LLVM IRs or assembly code. Fu et al. (2019) uses a LSTM model to generate code sketches for binary decompilation. In a second phase, these sketches are refined and corrected iteratively. Liang et al. (2021) trained a transformer model to decompile binaries to C. Szafraniec et al. (2022) leveraged LLVM IRs for source code translation, studying both IR inversion and IR-augmented pretraining methods. Similarly to our work, they consider IR decompilation to be a proxy objective for another task (code translation in their case).

4.3 Intermediate representations for static analysis

Static analysis is often performed on intermediate representations instead on the source code itself. A first immediate advantage is that one single IR can serve several language front-ends. This is also a design choice that is popular in compilers (see for example the LLVM project (Lattner and Adve, 2004)) which also perform static analysis, but for optimizations purposes rather than bugfinding. IR languages are much simpler than source languages: fewer syntactic constructs, and simpler language features. Analyzer designers save development and maintenance time with such an architecture.

4.4 Automatic program repair

Automatic program repair is a popular technique in IDE. For example, the Eclipse editor provides program repair for syntax errors. Our contribution deals with semantic error. For such errors, the dominant approach is based on testing (Perkins et al., 2009). When a program does not pass a test, a fixed program is searched and the success criterion is passing all the test suit. This approach can be directly driven at the source language level but it only observes a finite number of traces while static analysis can explore an infinite number by symbolic execution. A repair by static analysis is provided by the `ccheck` tool for C# programs (Logozzo and Ball, 2012). The static analysis is performed at the level of .NET bytecode language, which is a higher level representation than Infer IR, that deals with a strictly bigger set of frontend languages. The repaired properties they tackle are also of different nature because they require to modify a guard or adding an assignment in a linear block of instruction, while resource leak repair requires to build new control flow constructs like a `try ... finally ...` statement.

5 Conclusion and future works

Static analysis tools can flag potential issues in code, but they are often unable to propose suitable fixes. For instance, Infer produces an IR in which resource leaks can be found and fixed easily, but fixing the bug in the source code is difficult. We train a neural decompiler to retrieve the original source code from an Infer IR. This model generalizes to automatically fixed IRs, and it is able to propose proper fixes for up to 66.4% of the code snippets in our test dataset. We measure the semantic correctness of our proposed fixes using unit tests and show that the precision varies between 96.4% and 97.9%. We believe that our method is precise enough to propose automatic PR for bug fixes, or automatic fixes in IDEs.

In this work, we only used Infer IRs built from Java files that compile in isolation. Integrating Infer to build tools and running it on entire projects would allow us to train the decompiler on much larger datasets and to improve its performance. The lack of correctness guarantees frequently hinders the adoption of machine learning models for code repair. Assuming that the compiler and our fixes in IR space are correct, it would be possible to get such guarantees for our system. The source code we produce can be compiled to IR again, and compared to the fixed IR. In our current setting, our fixed IR are unnatural and often do not match the IR produced by compiling any source code. Hence, we would like to develop automated methods to compare IRs semantically Benton (2018) or even propose **semantic distances** between programs. We also plan to extend this method to other types of bugs flagged by Infer, such as buffer overflows, dead code, and null or dangling pointer dereferences.

REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2655–2668, 2021.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. Advances in Neural Information Processing Systems, 34:27865–27876, 2021.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. Proceedings of the ACM on Programming Languages, 3(OOPSLA):1–27, 2019.

Nick Benton. Semantic equivalence checking for HHVM bytecode. In Proc. of PPDP 2018, pages 3:1–3:8. ACM, 2018.

Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pages 60–70. IEEE, 2018.

Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In Proceedings of the 11th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2017, pages 1–11, 2017.

Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E Díaz Ferreyra. Vul4j: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In Proceedings of the 19th International Conference on Mining Software Repositories, pages 464–468, 2022.

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering, 2019.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311, 2022.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In International Conference on Learning Representations (ICLR), 2020.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. Commun. ACM, 62(8):62–70, 2019.

Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. arXiv preprint arXiv:2105.09352, 2021.

Thomas Durieux and Martin Monperrus. Introclassjava: A benchmark of 297 small and buggy java programs. 2016.

Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. arXiv preprint arXiv:1909.11556, 2019.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In Advances in Neural Information Processing Systems, pages 3703–3714, 2019.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In Thirty-First AAAI Conference on Artificial Intelligence, 2017.

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

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. arXiv preprint arXiv:1904.09751, 2019.

Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 388–398. IEEE, 2019.

René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In Proceedings of the 2014 international symposium on software testing and analysis, pages 437–440, 2014.

Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. arXiv preprint arXiv:1905.08325, 2019.

Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In Conference of the Association for Machine Translation in the Americas, pages 115–124. Springer, 2004.

Sophia D Kolak, Ruben Martins, Claire Le Goues, and Vincent Josua Hellendoorn. Patch generation with language models: Feasibility and scaling behavior. In Deep Learning for Code Workshop, 2022.

Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. arXiv preprint arXiv:2102.07492, 2021.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In Proc. of CGO'04, pages 75–88, San Jose, CA, USA, Mar 2004.

Ruigang Liang, Ying Cao, Peiwei Hu, Jinwen He, and Kai Chen. Semantics-recovering decompilation through neural machine translation. arXiv preprint arXiv:2112.15491, 2021.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 1–12. IEEE, 2019.

Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In Proc. of OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012, pages 133–146. ACM, 2012.

Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19), 2019. URL https://arxiv.org/abs/1901.06024.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics, pages 311–318. Association for Computational Linguistics, 2002.

Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Gregory T. Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. Automatically patching errors in deployed software. In Proc. of SOSP 2009, pages 87–102. ACM, 2009.

Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages, 2(OOPSLA):1–25, 2018.

Julian Aron Prenner and Romain Robbes. Automatic program repair with openai's codex: Evaluating quixbugs. arXiv preprint arXiv:2111.03922, 2021.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. Project CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. CoRR, abs/2105.12655, 2021. URL https://arxiv.org/abs/2105.12655.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. Advances in Neural Information Processing Systems, 33, 2020.

Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In Proceedings of the 15th international conference on mining software repositories, pages 10–13, 2018.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pages 1715–1725, 2015.

Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. arXiv preprint arXiv:2207.03578, 2022.

Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 180–182. IEEE, 2017.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Transactions on Software Engineering and Methodology (TOSEM), 28(4):1–29, 2019.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. arXiv preprint arXiv:1904.01720, 2019.

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, pages 5998–6008, 2017.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In Proceedings of the 25th international conference on Machine learning, pages 1096–1103, 2008.

Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 479–490. IEEE, 2019.

Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In International Conference on Machine Learning, pages 10799–10808. PMLR, 2020.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. Selfapr: Self-supervised program repair with test execution diagnostics. arXiv preprint arXiv:2203.12755, 2022.

A GENERATION EXAMPLES.

Original program	Decompiled program

Original program:

```java
final class UploadRequest {
    private static void writeFileToOutputStream(
                                    File file,
                                    OutputStream outputStream)
                                    throws IOException {
        FileInputStream inputStream = null;
        try {
          inputStream = new FileInputStream(file);
          byte[] buffer = new byte[4096];
          int bytesRead;

          while ((bytesRead = inputStream.read(buffer)) > 0) {
            outputStream.write(buffer, 0, bytesRead);
          }
          outputStream.flush();

        } finally {
          if (inputStream != null) {
            inputStream.close();
          }
        }
      }
}
```

Decompiled program:

```java
final class UploadRequest {
    private static final int BUFFER_SIZE = 4 * 1024 ;
    private static void writeFileToOutputStream(
                                    File file,
                                    OutputStream outputStream)
                                    throws IOException {
        FileInputStream inputStream = null;
        try {
          inputStream = new FileInputStream(file);
          byte[] buffer = new byte[BUFFER_SIZE];
          int bytesRead;
          while ((bytesRead = inputStream.read(buffer)) > 0) {
            outputStream.write(buffer, 0, bytesRead);
          }
          outputStream.flush();

        } finally {
          if (inputStream != null) {
            inputStream.close();
          }
        }
      }
}
```

Original program:

```java
static ArrayList<String> readListFromFile(File targetFile) {
    ArrayList<String> fileContents;
    try (ObjectInputStream inputStream = new ObjectInputStream(
                        new FileInputStream(targetFile))) {
      fileContents=(ArrayList<String>) inputStream.readObject();
      return fileContents;
    } catch (Exception e) {
      return new ArrayList<>();
    }
  }
```

Decompiled program:

```java
static ArrayList<String> readListFromFile(File targetFile) {
    try {
        ObjectInputStream inputStream = new ObjectInputStream(
                        new FileInputStream(targetFile));
        ArrayList<String>
        fileContents=(ArrayList<String>) inputStream.readObject();
        return fileContents;
    }
    catch (IOException e) {
        return new ArrayList<String>();
    }
    catch (ClassNotFoundException e) {
        return new ArrayList<String>();
    }
}
```

Original program:

```java
private static void addFileToZip(
                        File base,
                        String filePath,
                        ZipOutputStream outZipStream)
                        throws IOException {
    try (FileInputStream inputStream = new FileInputStream(
                           new File(base, filePath)); ) {
        byte[] buffer = new byte[BUFFER_SIZE_BYTES];
        outZipStream.putNextEntry(new ZipEntry(filePath));
        int bytes_read;
        while ((bytes_read = inputStream.read(buffer)) > 0) {
            outZipStream.write(buffer, 0, bytes_read);
        }
    } finally {
        outZipStream.closeEntry();
    }
}
```

Decompiled program:

```java
private static void addFileToZip(
                        File base,
                        String filePath,
                        ZipOutputStream outZipStream)
                        throws IOException {
    try {
        FileInputStream inputStream = new FileInputStream(
                            new File(base, filePath));
        try {
            byte[] buffer = new byte[BUFFER_SIZE_BYTES];
            outZipStream.putNextEntry(new ZipEntry(filePath));
            int bytes_read;
            while ((bytes_read = inputStream.read(buffer)) > 0) {
                outZipStream.write(buffer, 0, bytes_read);
            }
        }
        finally {
            outZipStream.closeEntry();
        }
    }
    finally {
    }
}
```

Figure 3: **Example decompilations** In this table, we highlight some examples of decompilations proposed by the decompiler. In this case, the input to the model is the IR of the original program in the left side of the table (without the repair step), and the output in the right side is the output of the model using greedy decoding. In these examples, we observe some style improvements in exception handling and variable assignment. The third example is a non-trivial decomposition of modern try-with-resources into multiple try finally statements.

```java
// Original function
public static void foo () throws IOException{
    FileOutputStream fos = new FileOutputStream(new File("/tmp/bar.txt"));
    fos.write(7);
}
```

```
# Infer IR
define Main.foo() : void {
  #node_0:
      jmp node_2
      .handlers node_1

  #node_1:
      jmp

  #node_4:
      jmp node_1
      .handlers node_1

  #node_2:
      n0 = __sil_allocate(<java::io::File>)
      n1 = java::io::File.<init>(n0, "/tmp/bar.txt")
      store &$irvar0 <- n0:*java::io::File
      jmp node_6
      .handlers node_4

  #node_6:
      n2 = __sil_allocate(<java::io::FileOutputStream>)
      n3:*java::io::File = load &$irvar0
      n4 = java::io::FileOutputStream.<init>(n2, n3)
      store &fos <- n2:*java::io::FileOutputStream
      jmp node_5
      .handlers node_4

  #node_5:
      n5:*java::io::FileOutputStream = load &fos
      n6:*java::io::FileOutputStream = load n5
      n7 = n5.java::io::FileOutputStream.write(7)
      jmp node_3
      .handlers node_4

  #node_3:
      jmp node_1
      .handlers node_4
}
```

```
# Fixed Infer IR
define Main.foo() : void {
  #node_0:
      jmp node_2
      .handlers node_1

  #node_1:
      jmp

  #node_4:
      n4:*java::io::FileOutputStream = load &fos
      n5 = n4.java::io::FileOutputStream.close()
      jmp node_1
      .handlers node_1

  #node_2:
      n0 = __sil_allocate(<java::io::File>)
      n1 = java::io::File.<init>(n0, "/tmp/bar.txt")
      store &$irvar0 <- n0:*java::io::File
      jmp node_6
      .handlers node_4

  #node_6:
      n2 = __sil_allocate(<java::io::FileOutputStream>)
      n3:*java::io::File = load &$irvar0
      n4 = java::io::FileOutputStream.<init>(n2, n3)
      store &fos <- n2:*java::io::FileOutputStream
      jmp node_5
      .handlers node_4

  #node_5:
      n6:*java::io::FileOutputStream = load &fos
      n7:*java::io::FileOutputStream = load n6
      n8 = java::io::FileOutputStream.write(n6, 7)
      n2:*java::io::FileOutputStream = load &fos
      n3 = n2.java::io::FileOutputStream.close()
      jmp node_3
      .handlers node_4

  #node_3:
      jmp node_1
      .handlers node_4
}
```

```java
// Fixed function
public static void foo () throws IOException{
    FileOutputStream fos = new FileOutputStream(new File("..."));
    try {
      fos.write(7);
    } finally {
      fos.close();
    }
}
```

Figure 4: **Resource leak fix in IR space.** For the original function is shown on top of the figure, we show the Infer IR on the left, and the automatically fixed Infer IR on the right. The lines added in the fixed IR are higlighted in blue. Using our decompiler on the Infer IR on the right, we obtain the fixed function on the bottom, which uses `finally` to safely close the resource.