IReEn: Iterative Reverse-Engineering of Black-Box Functions via Neural Program Synthesis

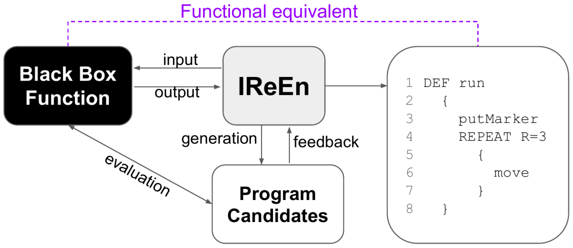
Hossein Hajipour[1], **Mateusz Malinowski** [2], **Mario Fritz**[1]
[1]CISPA Helmholtz Center for Information Security
[2] DeepMind
`{hossein.hajipour,fritz}@cispa.saarland`,
`mateuszm@google.com`

Abstract

In this work, we investigate the problem of revealing the functionality of a black-box agent. Notably, we are interested in the interpretable and formal description of the behavior of such an agent. Ideally, this description would take the form of a program written in a high-level language. This task is also known as *reverse engineering* and plays a pivotal role in software engineering, computer security, but also most recently in interpretability. In contrast to prior work, we do not rely on privileged information on the black box, but rather investigate the problem under a weaker assumption of having only access to inputs and outputs of the program. We approach this problem by iteratively refining a candidate set using a generative neural program synthesis approach until we arrive at a functionally equivalent program. We assess the performance of our approach on the Karel dataset. Our results show that the proposed approach outperforms the state-of-the-art on this challenge by finding a functional equivalent program in 78% of cases – even exceeding prior work that had privileged information on the black-box.

1 Introduction

Reverse-engineering (RE) is about gaining insights into the inner workings of a mechanism, which often results in the capability of reproducing the associated functionality. In our work, we consider a program to be a black-box function that we have no insights into its internal mechanism, and we can only interface with it through inputs and the program generated outputs. This is a desired scenario in software engineering (Lee et al., 2011; Fu et al., 2019), or security, which reverse-engineers, e.g., binary executables for analysis and for finding potential vulnerabilities (Kolbitsch et al., 2009; Yakdan et al., 2016). Furthermore, a similar paradigm is used to reverse-engineer the brain to advance knowledge in various brain-related disciplines (Markram, 2012) or seeking interpretation for reinforcement learning agents (Verma et al., 2018). More recently, similar principles have been applied to the machine learning domain to find internal information or copy black-box models like deep learning architectures (Tramèr et al., 2016; Oh et al., 2018; Orekondy et al., 2019).



```
DEF run
  {
    putMarker
    REPEAT R=3
      {
        move
      }
  }
```

Figure 1: An example of revealing the functionality of a black-box function using only input-output interactions.

Despite all of the progress in reverse-engineering the software and machine learning models, there are typical limitations in the proposed works. E.g., in the decompilation task, one of the assumptions is to have access to the assembly code of the black-box programs or other privileged information, which is a significant information leak about the black-box function. Furthermore, in deep learning, a common issue is that the reverse-engineered models usually are not represented in an interpretable and human-readable form.

In recent work, neural program synthesizers are employed to recover a functional and interpretable form of a black-box program that is generated based only on I/Os examples Bunel et al. (2018). On close inspection, however, it turns out, that these approaches also leverage privileged information, by relying on a biased sampling strategy of I/Os that was obtained under the knowledge of the black-box function.

In contrast to prior work, we propose an iterative neural program synthesis scheme which is the first to tackle the task of reveres-engineering in a black-box setting without any access to privileged information. Despite the weaker assumptions and hence the possibility to use our method broadly in other fields, we show that in many cases it is possible to reverse engineer functionally equivalent programs on the Karel dataset benchmark. We even achieve better results than prior work that has access to privileged information.

We achieve this by an iterative reverse-engineering approach. We query a black-box function using random inputs to obtain a set of I/Os, and refine the candidate set by an iterative neural program synthesis scheme. This neural program synthesis is trained with pairs of I/Os and target programs. To adapt our program synthesize to the domain of random I/Os we fine-tune our neural program synthesize using random I/Os and the corresponding target program.

To summarize the contributions of this work are as follow:

1. We propose an iterative neural program synthesizer scheme to reverse-engineer a functional equivalent form of the black-box program. To the best of our knowledge, this is the first approach that operates in a black-box setting without privileged information.
2. We proposed a functional equivalence metric in order to quantify progress on this challenging task.
3. We evaluate our approach on Karel dataset, where our approach successfully revealed the underlying programs of 78% of the black-box programs. Our approach outperforms prior work despite having access to less information due to weaker assumptions.

2 Related Work

Reverse-Engineering of programs. Decompilation is the task of translating a low-level program into a human-readable high-level language. Phoenix (Brumley et al., 2013) and Hex-Rays (Hex, 1998) are conventional decompilers. These decompilers are relying on pattern matching and hand-crafted rules, and often fail to decompile non-trivial codes with conditional structures. Fu et al. (2019); Katz et al. (2019) proposed a deep-learning-based approach to decompile the low-level codes in an end-to-end fashion. In the decompilation task, the main assumption is to have access to a low-level code of the program. However, in our approach, our goal is to represent a black-box function in a high-level program language only by relying on input-output interactions.

Reverse-Engineering of neural networks. Reverse-engineering neural network recently has gained popularity. Oh et al. (2018) proposed a meta-model to predict the attributes of the black-box neural network models, such as architecture and optimization process. Orekondy et al. (2019) investigate how to steal the functionality of the black-box model only based on image query interactions. While these works try to duplicate the functionality of a black-box function, in this work our goal is to represent the functionality of the black-box function in a human-readable program language.

Reverse-Engineering for interpretability. In another line of work, Verma et al. (2018); Bastani et al. (2018) proposed different approaches to have interpretable and verifiable reinforcement learning. Verma et al. (2018) designed a reinforcement learning framework to represent the policy network using human-readable domain-specific language, and Bastani et al. (2018) represent policy network by a training decision tree. Both of these works are designed for a small set of RL problems with a simple program structure. However, in our work, we consider reverse-engineer a wide range of programs with complex structures.

Program synthesis. Program synthesis is a classic task which has been studied since the early days of Artificial Intelligence (Waldinger and Lee, 1969; Manna and Waldinger, 1975). Recently there has been a lot of recent progress in employing a neural-networks-based approach to do the task of program synthesis. One type of these approaches called *neural program induction* involves learning a machine learning model to mimic the behavior of the target program (Graves et al., 2014; Johnson et al., 2017; Devlin et al., 2017a). Another type of approach is *neural program synthesis*, where the goal is to learn to generate an explicit discrete program in a domain-specific program language. Devlin et al. (2017b) proposed an encoder-decoder neural network style to learn to synthesize programs from input-output examples. Bunel et al. (2018) synthesizing Karel programs from examples, where they learn to generate program using a deep-learning-based model by leveraging the syntax constraints and reinforcement learning. Shin et al. (2018); Chen et al. (2019) leverage the semantic information of execution trace of the programs to generate more accurate programs. These works assume that they have access to the crafted I/O examples. However, in this work, we proposed an iterative program synthesis scheme to deal with the task of black-box program synthesis, where we only have access to the random I/O examples.

3 Problem Overview

In this section, we formulate the problem description and our method. We base our notation on (Bunel et al., 2018; Shin et al., 2018; Chen et al., 2019).

Program synthesis. Program synthesis deals with the problem of deriving a program in a specified programming language that satisfies the given specification. We treat input-output pairs $I/O = \left\{(I^k, O^k)\right\}_{k=1}^{K}$ as a form of specifying the functionality of the program. This problem can be formalized as finding a solution to the following optimization problem:

$$\arg\min_{p\in\mathcal{P}} \quad \Omega(p) \tag{1}$$

$$\text{s.t.} \quad p(I^k) = O^k \quad \forall k \in \{1, \dots, K\} \tag{2}$$

where $\mathcal{P}$ is the space of all possible programs written in the given language, and Ω is some measure of the program. For instance, Ω can be a cost function that chooses the shortest program.

The situation is illustrated in Figure 2. For many applications – also the one we are interested in – there is a true underlying black-box program that satisfies all the input-output pairs. As most practical languages do not have a unique representation for certain functionality or behavior, a certain set of functionally equivalent programs will remain indistinguishable even given an arbitrary large number of input-output observations and respective constraints in our optimization problem. Naturally, by adding more constraints, we obtain a nested constraint set that converges towards the feasible set of functionally equivalent programs.

Program synthesis with privileged information. Recent works implicitly or explicitly incorporates insider information on the function to reverse-engineer. This can come in the form of a binary of the compiled code or an informed sampling strategy of the input-output pairs. It turns out that the majority of recent research implicitly uses privileged information via biased sampling scheme in terms of *crafted* specifications (Bunel et al., 2018; Chen et al., 2019; Pattis, 1981; Shin et al., 2018). Note that in order to arrive at these specifications, one has to have access to the program P under the question as they are designed to capture e.g. all branches of the program. We call these crafted specifications *crafted I/Os* and will investigate later in detail how much information they leak about the black-box program.

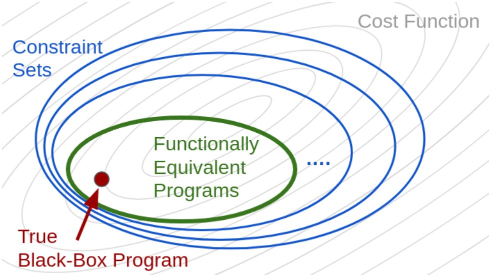


Figure 2: Illustration of the optimization problem, functional equivalence, and feasible sets w.r.t. nested constraint sets.

Black-Box program synthesis. In our work, we focus on a black-box setting, where no such side or privileged information is available. Hence, we will have to defer initially to randomly generate K

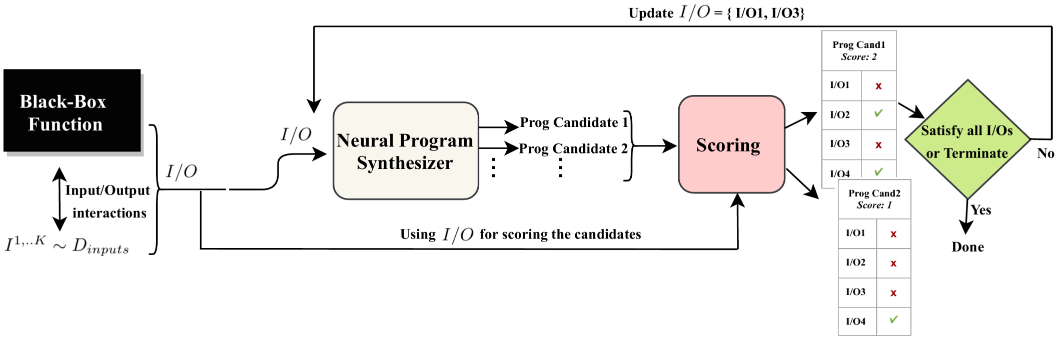

Update I/O = { I/O1, I/O3}
Black-Box Function
Input/Output interactions
$I^{1,..K} \sim D_{inputs}$
I/O
I/O
Neural Program Synthesizer
Prog Candidate 1
Prog Candidate 2
Scoring
Using I/O for scoring the candidates
Prog Cand1
Score: 2
I/O1 ✗
I/O2 ✓
I/O3 ✗
I/O4 ✓
Prog Cand2
Score: 1
I/O1 ✗
I/O2 ✗
I/O3 ✗
I/O4 ✓
Satisfy all I/Os or Terminate
No
Yes
Done

Figure 3: Overview of the proposed iterative neural program synthesis approach.

inputs $\{I^k\}_{k=1}^K$ and next query the program p to obtain the corresponding outputs $\{O^k\}_{k=1}^K$. Such generated input-output pairs become our specification that we use to synthesize programs. Note that, unlike the previous setting, here we take advantage of querying the black-box program p in an active way, even though the whole procedure remains automatic. To generate random inputs we follow the procedure proposed by Bunel et al. (2018). We call the obtained I/Os in the black-box setting *random I/Os*. It turns out (as we will also show in our experiments), that indeed such random, uninformed input queries yield significantly less information than the *crafted I/Os* used in prior work. Hence, to arrive at an effective and black-box approach, in the following we propose an iterative reverse-engineering scheme, that gradually queries more relevant inputs.

4 IReEn: Iterative Reverse-Engineering of Black-Box Functions

Reverse-engineering a black-box function and representing it in a high-level language is a challenging task. The main reason is that we can only interact with the black-box function using input-output examples. In addition, solving the above constraint optimization problem is intractable. Therefore, in the following, we relax the optimization problem to a Bayesian inference problem and show how to iteratively incorporate additional constraints in order to arrive at a functional equivalent program with respect to the black-box function.

Figure 3 provides an overview of our iterative neural program synthesis scheme to reverse-engineer the given black-box function. In the first step, we obtain the I/Os by querying the black-box function using random inputs drawn from a distribution of inputs. We condition the neural program synthesizer on the obtained I/Os. Neural program synthesizer outputs the potential program candidate(s), and then we use a scoring system to score the generated candidates. For example, in this figure "program candidate 1" satisfied two out of four sample I/Os, so its score will be 2. If the best candidate does not cover all of the I/Os, we select a subset of I/Os which were not covered by the best candidate program to condition them on program synthesizer for the next iteration.

4.1 Finding Programs given Input-Output Constraints

Even for a small set of input-output constraints, finding the feasible set of programs that satisfies these I/Os is not tractable due to the discrete and compositional nature of programs. We approach this challenging problem by relaxing the constraint optimization problem to a Bayesian Inference problem. The scoring function Ω is modeled by a prior $\lambda(\hat{p})$ and the feasible set is represented by a uniform distribution $\lambda(\hat{p}|\{(I_i^k, O_i^k)\}_{k=1}^K)$ over the feasible set. In this way, samples of the posterior are solutions to the constraint optimization problem. In order to train such a generative model, we directly optimize the Neural Program Synthesis approach based on Bunel et al. (2018) to fit this posterior. This is a conditional generative model that samples candidate programs by conditioning on the input-output information.

$$\hat{P} \sim \Psi(I/O). \tag{3}$$

Where $\hat{P}$ is a set of sampled solutions that are program candidate(s) $\{\hat{p}_1, ..., \hat{p}_C\} \in \hat{P}$ and $C \geq 1$.

In detail, we train this recurrent encoder and decoder for program synthesis on a set of ground-truth programs $\{p_i\}_i$ and specifications $\{I/O_i\}_i$. Each specification is a set of K pairs $I/O_i = \{(I_i^k, O_i^k)\}_{k=1}^K$ where the program needs to be consistent with, that is, $p_i(I_i^k) = O_i^k$ for all $k \in \{1, \ldots, K\}$. In our work, we pre-train the program synthesis proposed by Bunel et al. (2018), where they use encoder-decoder neural networks to generate the desired program given input-output specifications. Note that the synthesizer is dependent on the input specification, that is, different I/O_a and I/O_b may produce different programs through the synthesis, i.e., $\Psi(I/O_a) = p_a$ and $\Psi(I/O_b) = p_b$. For a detailed discussion, e.g. of the I/O encodings, we refer to Bunel et al. (2018).

4.2 Sample Rejection Strategy

Naturally, we expect approximation errors of the optimization problem by the generative model. Two main sources of error are (1) challenges to approximate the discontinuous target distribution (2) only a limited (and fixed) number of constraints can be incorporated in the conditional generative model. In order to correct for these errors, we follow up with a sample rejection stage based on a scoring of the generated program candidates. We use random I/Os obtained from interacting with black-box function to evaluate the generated programs, and score them based on the number of the I/Os which were covered by the programs (See Figure 3). While in principle, any failed I/O should lead to rejecting a candidate, empirically, we find that keeping the highest scoring samples turns out to be advantageous and prevents situations where no candidates would remain.

4.3 Iterative Refinement

We are still facing two major issues: (1) As we have motivated before and also our experiments will show, querying for certain I/O pairs is more informative than others. Hence, we seek an iterative approach that yields more informative queries to the black box. (2) The conditional generative model only takes a small and fixed number of constraints, while it is unclear which constraints to use in order to arrive at the "functional equivalent" feasible set.

Similar problems have been encountered in constraint optimization, where *column generation algorithms / delayed constraint generation* techniques have been employed to deal with large number of constraints (Ford Jr and Fulkerson, 1958). Motivated by these ideas, we propose an iterative strategy, where we condition in each step on a set of violated constraints that we find.

In detail, we present the algorithm of the proposed method in Algorithm 1. Iterative synthesis function takes synthesizer Ψ and a set of I/Os (line 1). In line 2 we initial the s_{best} to zero. Note that we use p_{best} to store the best candidate, and s_{best} to store the score of the best candidate. In the iterative loop, we first condition the program synthesizer on the given I/O set to get the program candidates $\hat{P}$ (line 5). Then we call *Scoring* function to score the program candidates in line 6. The scoring function returns the best program candidate, the score of that candidate, and the new set of I/O where the new I/Os are the one which were not satisfied by $\hat{p}_{best}$. Note that $\hat{p}_{best}$, and $\hat{s}_{best}$ store the best candidate and the score of it for the current iteration. Then at line 7, we check if $\hat{s}_{best}$ for the current iteration is larger than the global score s_{best}, if the condition satisfies we update the global p_{best}, and s_{best} (line 8-9). In line 12 we return the best candidate p_{best} after searching for it for n iterations.

4.4 Fine-tuning

The goal of synthesizer Ψ is to generate a program for the given I/Os, so it is not desirable to generate a program that contains not-used statements (e.g. a while statement which never hit by the given I/Os). However, in the black-box setting, we only have access to the random I/Os, and there is no guarantee if these I/Os represent all details of the black-box program. So the synthesizer might need to generate a statement in the program which was not represented in the given I/Os. The question is how we can have a synthesizer that makes a balance between these two contradictory situations. To address this issue, we first train synthesizer Ψ on the crafted I/Os and then fine-tune it on the hard examples of random I/Os. We consider a random I/Os as a hard example for the Ψ if the generated program by Ψ does not satisfy all of the given random I/Os. We get the data for fine-tuning by feeding random I/Os to the Ψ which were trained on crafted I/Os, and we pair the hard examples of random I/Os with the target programs.

5 Experiments

In this section, we show the effectiveness of our proposed approach for the task of black-box program synthesis. We consider Karel dataset (Devlin et al., 2017a; Bunel et al., 2018) in a strict black-box setting, where we can only have access to I/Os by querying the black-box functions without any privileged information or informed sampling scheme.

Algorithm 1: Iterative Algorithms

```
Function IterativeSynthesis(Ψ, I/O):
    $s_{best} = 0$ // To keep the best score.
    n = $constant$ // e.g.n=10
    for $i \leftarrow 1$ to $n$ do
        $\hat{P} = \Psi(I/O)$
        $\hat{p}_{best}, \hat{s}_{best}, I/O = Scoring(\hat{P})$
        if $s_{best} < \hat{s}_{best}$ then
            $p_{best} = \hat{p}_{best}$
            $s_{best} = \hat{s}_{best}$
        end
    end
    return $p_{best}$
End Function
```

5.1 The Karel Task and Dataset

To evaluate our proposed approach, we consider Karel programming language. Karel featured a robot agent in a grid world, where this robot can move inside the grid world and modify the state of the world using a set of predefined functions and control flow structures. Recently it has been used as a benchmark in several neural program synthesis works (Bunel et al., 2018; Shin et al., 2018; Chen et al., 2019). Using control flow structures such as condition and loop in the grammar of Karel makes this DSL a challenging language for the task of program synthesis.

Bunel et al. (2018) defined a dataset to train and evaluate neural program synthesis approaches by randomly sampling programs from the Karel's DSL. In this dataset, for each program, there is 5 I/Os as specification, and one is the held-out test sample. In this work, we consider the Karel's programs as black-box agent's task, and our goal is to reveal the underlying functionality of this black-box function by solely using input-output interactions. This dataset contains 1,116,854 pairs of I/Os and programs, 2,500 for validations, and 2,500 for testing the models. Note that, to fine-tune the synthesizer to the domain of random I/Os, we used 100,000 pairs of random I/Os and target program for training and 2,500 for validation.

5.2 Training and Inference

We train the neural program synthesizer using the Karel Dataset. To train this synthesizer we employ the neural networks architecture proposed by Bunel et al. (2018), and use that in our iterative refinement approach as the synthesizer. Note that, to fine-tune the synthesizer model on random I/Os we use Adam optimizer (Kingma and Ba, 2015) and the learning rate 10^{-5}, we fine-tune the synthesizer model for 10 epochs. During inference, we use beam search algorithm with beam width 64 and select top-k program candidates.

5.3 Functional Equivalence Metric

In Bunel et al. (2018); Shin et al. (2018), two metrics have been used to evaluate the trained neural program synthesizer. 1. Exact Match: A predicted program is an exact match of the target if it is the same as the target program in terms of tokens. 2. Generalization: A predicted program is a generalization of the target if it satisfies the I/Os of the specification set and the held-out example. Both of these metrics have some drawbacks. A predicted program might be functionally equivalent to the target program but not be the exact match. On the other side, a program can be a generalization of the target program by satisfying a small set of I/Os (In Bunel et al. (2018) 5 I/Os has been used as specification and 1 I/O is considered as held-out). However, it might not cover a larger set of I/Os for that target program. To overcome this issue, in this work we proposed the Functional Equivalence metric, where we consider a predicted program as a functional equivalence of the target program if it covers a large set of I/Os which were not been used as the specification in the synthesizing time. To get the set of I/Os, we generate the inputs randomly and query the program to get the outputs. We check if these inputs hit all of the branches of the target program.

Models	Generalization		Functional		Exact Match	
	top-1	top-50	top-1	top-50	top-1	top-50
Random I/Os	57.12%	71.48%	49.36%	63.72%	34.96%	40.92%
Random I/Os + FT	64.72%	77.64%	55.64%	70.12%	39.44%	45.4%
Random I/Os + IReEn	76.20%	85.28%	61.64%	73.24%	40.95%	44.99%
Random I/Os + FT + IReEn	**78.96%**	**88.39%**	**65.55%**	**78.08%**	**44.51%**	**48.11%**
Crafted I/Os (Bunel et al. (2018))	73.12%	86.28%	55.04%	68.72%	40.08%	43.08%

Table 1: Top: Results of performance comparison of our approach in different settings using random I/Os for black-box program synthesis. Random I/Os mean that we use randomly obtained I/Os in the black-box setting, FT refers to fine-tuned model, and IReEn denote to our iterative approach. Bottom: Results of Bunel et al. (2018) when we use crafted I/Os. top-1 denote the results for the most likely candidate, and top-50 denote the results for 50 most likely candidates.

5.4 Evaluation

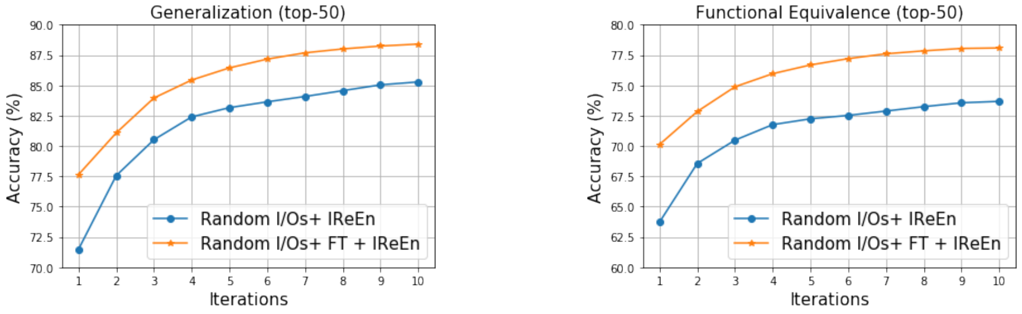
We investigate the performance of our approach in different settings to do the task of black-box program synthesis. To evaluate our approach we query each black-box program in the test set with 50 valid inputs to get the corresponding outputs. Using the obtained 50 I/Os we synthesize the target program, where we use 5 out of 50 I/Os to conditions on the synthesizer and use 50 I/Os to score the generated candidate and find the best one based on sample rejection strategy. In our iterative approach in each iteration, using sample rejection strategy we find a new 5 I/Os among the 50 I/Os to condition on the synthesizer for the next iteration. To evaluate the generated programs, in addition to generalization and exact match accuracy, we also consider our proposed metric called Functional Equivalence (subsection 5.3). To compute the functional equivalency we use 100 I/Os which were not seen by the model. If the generated program satisfies all of 100 I/Os we consider it as a program which functionally equal to the target program. In all of the results top-k means that we use the scoring strategy to find the best candidate among the "k" top candidates. For computing the results for all of the metrics we evaluate the best candidate among top-k.

Comparison to baseline and ablation. Table 1 shows the performance of our approach in different settings in the top, and the results of the neural program synthesizer proposed by Bunel et al. (2018) in the bottom. These results show that when we only use random I/Os (first row), there is a huge drop in the accuracy in all of the metrics in comparison to the results of crafted I/Os. However, when we fine-tune the synthesizer the results improve in all of the metrics, especially for the top-1 and top-50 functional equivalence accuracy. Furthermore, when we use our iterative approach for 10 iterations with the fine-tuned model (fourth row), we can see that our approach outperforms even the crafted I/Os in all of the metrics. For example, it outperforms crafted I/Os in functional equivalence and exact match metric by a large margin, 9%, and 5% respectively for top-50 results.

Importance of the crafted I/Os. In Table 1 in the top first row (Random I/Os) we use random I/Os to condition on the synthesizer, and in the bottom (Crafted I/Os) we use crafted I/Os to condition on the same synthesizer. These results show that using random I/Os on the same synthesizer leads to 15%, and 5% drops in the results for top-50 generalization and functional accuracy respectively. Based on these results we can see that random I/Os contain significantly less information about the target program in comparison to the crafted I/Os.

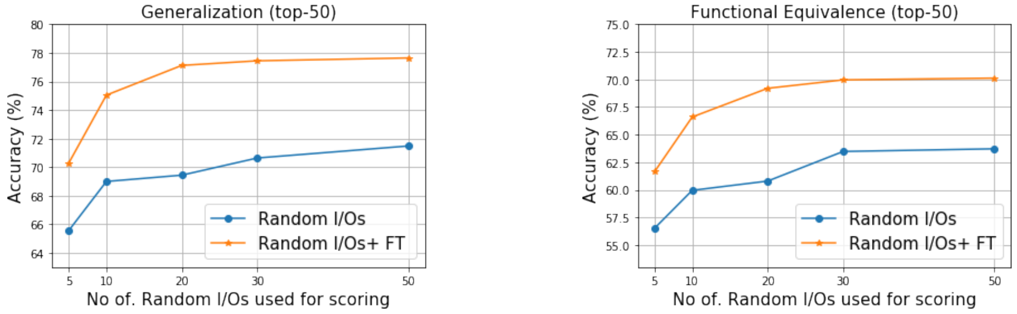
Effectiveness of iterative refinement. Figure 4a and Figure 4b show the effectiveness of our proposed iterative approach in 10 iterations. In these figures, x and y axis refer to the number of iterations and the accuracy respectively. Figure 4a shows the generalization accuracy for top-50, and in Figure 4b we can see the results of functional equivalence metric for top-50. In these figures, we provide results with and without fine-tuning the synthesizer. Here we can see the improvement of the generalization and functional equivalence accuracy over the iterations, we have a margin of 7% improvement in the functional equivalence accuracy for "Random I/Os + FT + IReEn" setting after 10 iterations. In other words, these results show that we can search for better random I/Os and program candidates by iteratively incorporate additional constraints.

Effectiveness of number of I/Os for sample rejection strategy. In our approach in order to choose one candidate among all of the generated program candidates, we consider a sample rejection strategy. To do that, we use the random I/Os to assign a score to the generated candidates based on the number of satisfied random I/Os. Finally, we consider the candidate with the higher score as the best candidate and reject the rest. Figure 5a and Figure 5b show the effect of using the different numbers of random I/Os on scoring the candidates and finding the best program candidate. x and y axis in these figures refer to the number of random I/Os, and accuracy of our approaches with and without fine-tuning. In Figure 5a we show the generalization accuracy and in Figure 5b we provide functional equivalence results. These figures show that by using more random I/Os in the scoring strategy we can find more accurate programs that result to gain better performance in terms of generalization and functional equivalence accuracy. In other words, with using more random I/Os for scoring the candidates we can capture more details of the black-box function, and find the best potential candidate among the generated one.



(a) (b)

Figure 4: (a) Generalization accuracy after each iteration for "Random I/Os + IReEn" and "Random I/Os + FT + IReEn". (b) Functional equivalence accuracy after each iteration for "Random I/Os+ IReEn" and "Random I/Os + FT + IReEn". Note that, Random I/Os means that we use randomly obtained I/Os, FT denotes to the fine-tuned model, and IReEn refers to our iterative approach.



(a) (b)

Figure 5: (a) Generalization accuracy after using the different numbers of random I/Os in scoring strategy for "Random I/Os" and "Random I/Os + FT". (b) Functional equivalence accuracy after using the different numbers of random I/Os in scoring strategy for "Random I/Os" and "Random I/Os + FT". Note that, Random I/Os means that we use randomly obtained I/Os, FT denotes to the fine-tuned model, and IReEn refers to our iterative approach.

6 Conclusion

In this work, we propose an iterative neural program synthesis scheme to reverse-engineer the black-box functions and represent them in a high-level program. In contrast to previous works, where they have access to privileged information, in our problem setting, we only rely on the input-output interactions. To tackle the problem of reverse-engineering the black-box function in this challenging setting, we employ a neural program synthesizer in an iterative scheme. Using this iterative approach we search for the best program candidate in each iteration by conditioning the synthesizer on a set of violated constraints. Our evaluation on Karel dataset demonstrates the effectiveness of our proposed approach in finding functional equivalence programs. Besides this, the provided results show that our proposed approach even outperforming the previous work that uses privileged information to sample input-output examples.

References

Hex-rays, 1998. URL `https://www.hex-rays.com/products/ida/`.

O. Bastani, Y. Pu, and A. Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *NeurIPS*, 2018.

D. Brumley, J. Lee, E. J. Schwartz, and M. Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX*, 2013.

R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR*, 2018.

X. Chen, C. Liu, and D. Song. Execution-guided neural program synthesis. In *ICLR*, 2019.

J. Devlin, R. R. Bunel, R. Singh, M. Hausknecht, and P. Kohli. Neural program meta-induction. In *NIPS*, 2017a.

J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. In *ICML*, 2017b.

L. R. Ford Jr and D. R. Fulkerson. A suggested computation for maximal multi-commodity network flows. *Management Science*, 1958.

C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao. Coda: An end-to-end neural program decompiler. In *NeurIPS*, 2019.

A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv*, 2014.

J. Johnson, B. Hariharan, L. Van Der Maaten, J. Hoffman, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick. Inferring and executing programs for visual reasoning. In *CVPR*, 2017.

O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav. Towards neural decompilation. *arXiv*, 2019.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX*, 2009.

J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. 2011.

Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis. *Artificial intelligence*, 1975.

H. Markram. The human brain project. *Scientific American*, 2012.

S. J. Oh, B. Schiele, and M. Fritz. Towards reverse-engineering black-box neural networks. In *ICLR*. 2018.

T. Orekondy, B. Schiele, and M. Fritz. Knockoff nets: Stealing functionality of black-box models. In *CVPR*, 2019.

R. E. Pattis. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc., 1981.

E. C. Shin, I. Polosukhin, and D. Song. Improving neural program synthesis with inferred execution traces. In *NeurIPS*. 2018.

F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *USENIX*, 2016.

A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri. Programmatically interpretable reinforcement learning. In *ICML*, 2018.

R. J. Waldinger and R. C. Lee. Prow: A step toward automatic program writing. In *IJCAI*, 1969.

K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Security and Privacy (SP)*, 2016.

Appendix

Here we provide more details and information about our proposed work and the Karel dataset. In appendix A, we provide more details of the Karel programming language which were used to create the Karel dataset. Appendix B contains the neural network architecture and the training details of the neural program synthesizer. We discuss the details of the functional equivalence metric and show examples of functional equivalent programs in appendix C. In appendix D, we demonstrate the results of exact match accuracy to further investigate the effectiveness of the iterative refinement and the sample rejection strategy.

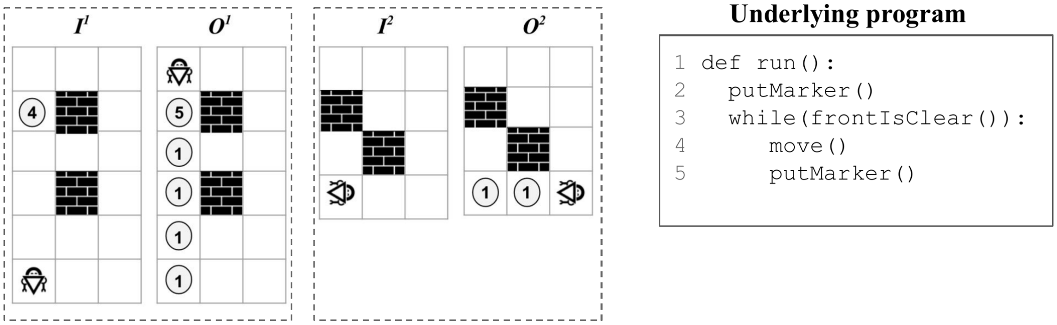
Appendix A More Details of the Karel Task

We use the Karel dataset to evaluate our proposed approach. This dataset was created based on Karel programming language (Pattis, 1981). Figure 6 shows the grammar specification of this programming language (Bunel et al., 2018). In this grammar, we can see the details of the Karel programming language, including different statements, conditions, and actions. This figure shows that Karel consists of control flow structures such as conditionals and loops, which make this programming language a challenging and complex task. Figure 7 demonstrates an example of the Karel task with two I/Os and the corresponding program.

To represent the Karel gird world (inputs and outputs), we use a tensor of size $16 \times H \times W$. Here we consider each grid world has a maximum size $16 \times 18 \times 18$. In this tensor representation, each cell in the grid is represented as a 16-dimensional vector. This 16-dimensional vector corresponds to the features described in Table 2.

```
  Prog p    :=  def run() : s
  Stmt s    :=  while(b) : s | repeat(r) : s | s1; s2 | a
             |  if(b) : s | ifelse(b) : s1 else : s2
  Cond b    :=  frontIsClear() | leftIsClear() | rightIsClear()
             |  markersPresent() | noMarkersPresent() | not b
Action a    :=  move() | turnRight() | turnLeft()
             |  pickMarker() | putMarker()
  Cste r    :=  0 | 1 | ··· | 19
```

Figure 6: The grammar for the Karel programming language.



Underlying program

```
def run():
  putMarker()
  while(frontIsClear()):
     move()
     putMarker()
```

Figure 7: Example of two I/Os of a Karel task with the corresponding underlying program. The robot is Karel, the brick walls represent obstacles, and markers are represented with circles.

Agent facing North
Agent facing South
Agent facing West
Agent facing East
Obstacle
Grid boundary
1 marker
2 marker
3 marker
4 marker
5 marker
6 marker
7 marker
8 marker
9 marker
10 marker

Table 2: Grid representation of each cell in the Karel world.

Appendix B Details of the Neural Program Synthesizer

One of the components of our iterative synthesizer scheme is a neural program synthesizer (Subsection 4.1 of the paper). To train this synthesizer we follow the design of the proposed work by Bunel et al. (2018). Here, we train an encoder-decoder architecture to generate the programs given the input-output examples. In this section, we provide details of the neural network architecture and the training procedure of the program synthesizer.

B.1 Neural Network Architecture

The encoder-decoder program synthesizer contains an encoder to encode the given I/O to a feature vector and a decoder that maps the given feature vector(s) to a set of program candidates. In particular, we use a convolutional neural network to encode the given I/O to a 512-dimensional vector. In the decoder, we have a 2-layer LSTM with a hidden size 256. Using this neural architecture, the decoder generates the candidate programs for the given encoded vectors of the I/Os.

B.2 Training Details

To train the neural program synthesizer we use Adam optimizer (Kingma and Ba, 2015), and the learning rate 10^{-4}. We train the model for 100 epochs with a batch size of 128.

Appendix C More Descriptions of the Functional Equivalence Metric

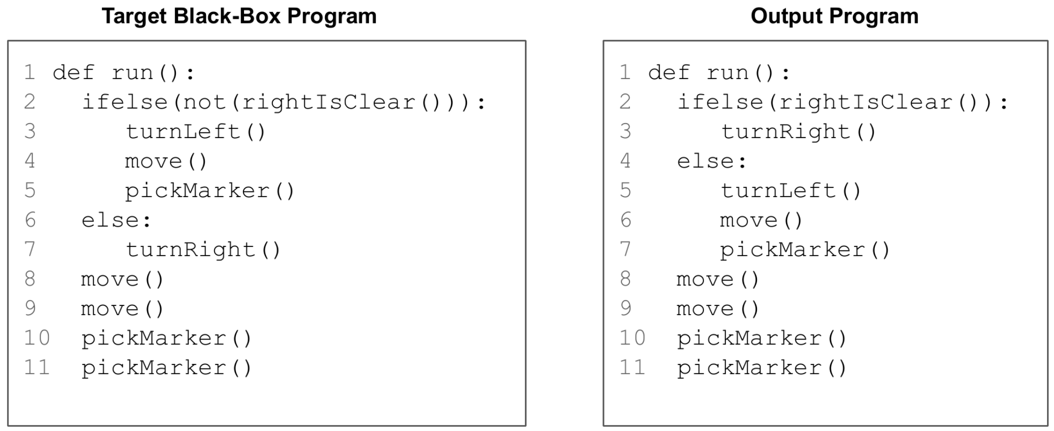
In our work, we propose a new metric to evaluate the program synthesizers. We called this metric Functional Equivalence, where we consider the predicted program is functionally equivalent to the target program if it covers a large set of I/Os. We provide the details of this metric in subsection 5.3 of the paper.

C.1 Details and Parameter

To compute the results for functional equivalency we use 100 I/Os which were not seen by the program synthesizer. We get these I/Os by generating random inputs and query the ground truth program to get the corresponding outputs (Section 5.3 of the paper). The predicted program is functionally equivalent to the ground truth program if it satisfies all of 100 I/Os. In our experiments, we found that with using more I/Os the more predicted programs we discover to not be functionally equivalent to the target programs. We found 100 number of I/Os as a point where the number of functionally equivalent programs stay stable in the evaluations.

C.2 Examples of Functional Equivalence Programs

Here we provide examples of functional equivalent programs. We found these examples using the results of functional equivalence accuracy. In Figures 8 and 9 we provide two examples of functional equivalent programs, which are not exact match of each other. In these figures, the left box shows the target black-box program, and the right box shows the output program, which was predicted by our iterative program synthesizer approach. In Figure 8, we can see there is a difference between the target black-box program and the output program at lines 2 to 7. However, this difference is only a swap in the *ifelse* condition and the corresponding functions which were used in the *ifelse* statement, so it does not affect the functionality of the programs. Figure 9 show another example of functional equivalent programs. In this figure, we can see that in line 2 of the target program there is a *repeat* statement that was not used in the output program. However, when we turn an agent for four times, the agent's direction will remain the same, so turning the agent in the same place for five times is equal to turning the agent for one time, and we have *trungRight()* function in line 4 of the output program, so having the *repeat* in target program does not affect the functional equivalency of these two programs.



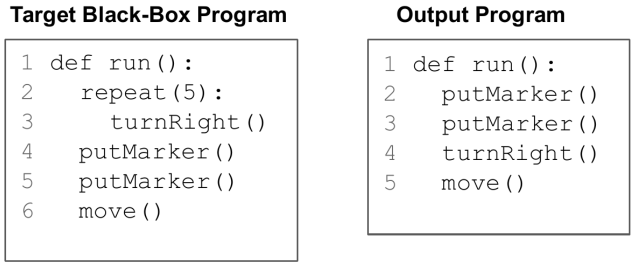
Target Black-Box Program

```
1  def run():
2    ifelse(not(rightIsClear())):
3       turnLeft()
4       move()
5       pickMarker()
6    else:
7       turnRight()
8    move()
9    move()
10   pickMarker()
11   pickMarker()
```

Output Program

```
1  def run():
2    ifelse(rightIsClear()):
3       turnRight()
4    else:
5       turnLeft()
6       move()
7       pickMarker()
8    move()
9    move()
10   pickMarker()
11   pickMarker()
```

Figure 8: Example of functional equivalent programs. Left: Target black-box program from the test set. Right: Output of our iterative program synthesis approach.



Target Black-Box Program

```
1 def run():
2   repeat(5):
3     turnRight()
4   putMarker()
5   putMarker()
6   move()
```

Output Program

```
1 def run():
2   putMarker()
3   putMarker()
4   turnRight()
5   move()
```

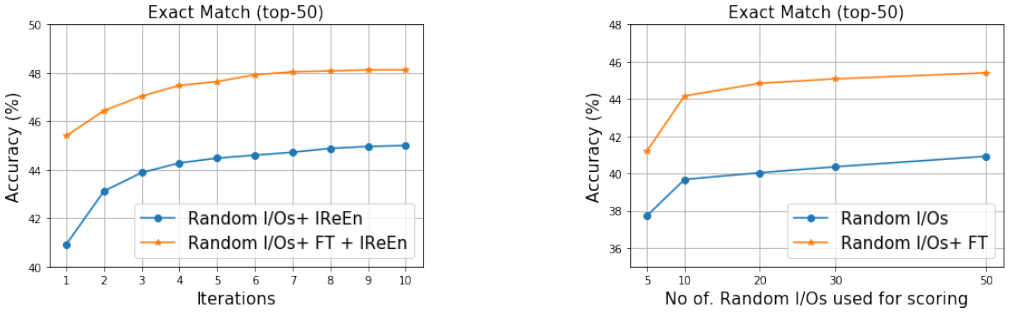
Figure 9: Example of functional equivalent programs. Left: Target black-box program from the test set. Right: Output of our iterative program synthesis approach.

Appendix D Exact Match Results

In our evaluations, we investigate the effect of iterative refinement and also the effect of using the different numbers of I/Os for the sample rejection strategy. In these experiments, we show the results for generalization accuracy and functional equivalence accuracy. Here, we provide results of exact match accuracy to see the effect of iterative refinement and sample rejection strategy in generating the programs which are the exact match of the target black-box programs.

Figure 10a shows the exact match accuracy for our proposed iterative approach in 10 iterations. In this figure, x and y axis refer to the number of iterations and the accuracy respectively. This figure demonstrates the exact match accuracy for top-50 with and without fine-tuning the neural program synthesizer. In Figure 10a, we can see the improvement of the exact match accuracy over the iterations, which shows the effectiveness of our iterative approach in searching for a better set of random I/Os to condition on the synthesizer.

Figure 10b shows the results of exact match accuracy for using the different numbers of random I/Os to score the program candidates. In this figure, x and y axis refer to the number of random I/Os, and the accuracy respectively. In Figure 10b, we provide results with and without fine-tuning the synthesizer for top-50 program candidates. This figure shows that by using more I/Os in the scoring strategy we can gain better performance in terms of exact match accuracy. For example, we can see that our approaches achieve the best exact match accuracy when we use 50 random I/Os to score the generated candidates. This shows that by using more random I/Os for scoring strategy we can find the best potential candidate among the generated one with higher accuracy.



(a) (b)

Figure 10: (a) Exact match accuracy after each iteration for "Random I/Os + IReEn" and "Random I/Os + FT + IReEn". (b) Exact match accuracy after using different numbers of random I/Os in scoring strategy for "Random I/Os" and "Random I/Os + FT". Note that, Random I/Os means that we use randomly obtained I/Os, FT denotes to the fine-tuned model, and IReEn refers to our iterative approach.