OVERFITTING IN PROGRAM REPAIR AND SYNTHESIS

XIANG GAO

NATIONAL UNIVERSITY OF SINGAPORE

2021

OVERFITTING IN PROGRAM REPAIR AND SYNTHESIS

XIANG GAO

(*B.S., Shandong University*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2021

Advisor:
Professor Abhik Roychoudhury

Examiners:
Associate Professor Wei Ngan Chin
Associate Professor Ilya Sergey
Professor Martin C. Rinard, Massachusetts Institute of Technology

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Xiang Gao

September 2021

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my PhD advisor, Prof. Abhik Roychoudhury for continuously encouraging and supporting me in the past several years. I learned almost everything about research and software engineering from him. Abhik always taught me what valuable research is, how to think about research problems and how to do influential research. His wise guidance extended beyond my research to my focus, communication, vision, ideas, and attitudes towards challenges. Through his invaluable advice, I learned so much about being professional.

I would like to thank my research intern advisors Sumit Gulwani, Nachiappan Nagappan, and Mukul R. Prasad for giving me much advice in terms of academics and industry. Beyond research papers, they gave a view of how valuable research can become products and make real impacts.

I would like to thank my undergraduate advisor Prof. Lei Ju. Lei taught me what research is and took me to the academic path. Without the guidance from him, I will not get a chance to be here and write this dissertation. Further, Lei suggested Abhik as my PhD advisor, and Abhik gave me exactly what Lei has described and even more. I would also like to thank my teachers Qingrong Sun and Jinfang Wang, who taught me how to speak, how to think, how to learn, and more importantly, how to be a human being. They left the deepest impression on me at my young age.

I am thankful to Wei Ngan Chin, Martin C. Rinard, and Ilya Sergey for taking time out of their busy schedule to serve in my thesis committee.

I would specially thank Sergey Mechtaev and Shin Hwei Tan. Both of them set examples for me in terms of focus, thinking, and productivity. At the early stage of my PhD, Sergey and Shin Hwei always patiently answer my research and technical questions, gently point out my mistakes and kindly help me improve my thinking, writing, and communicating.

I would also like to thank Gregory J. Duck, Arjun Radhakrishna, Ripon K. Saha, and Gustavo Soares for the collaborative works during my PhD journey. It is a great pleasure to work with them and discuss research questions. Their professionalism, passion, and kindness inspire me a lot. I also thank my collaborators Zhen Dong, Ruyi Ji, Yannic Noller, Ridwan Shariffdeen, and Bo Wang. Without them, I would

not achieve much alone.

Further, I would like to thank my friends and lab mates Umair Z. Ahmed, Abhijeet Banerjee, Marcel Böohme, Xiaoyu Du, Zhiyu Fan, Fuli Feng, Mingyuan Gao, Yang Hu, Yiming Liu, Yu Liu, Yunshan Ma, Ruijie Meng, Manh-Dung Nguyen, Suyi Ong, Van-Thuan Pham, Jyoti Prakash, Shiqi Shen, Abhishek Tiwari, Edwin Lesmana Tjiong, Guanhua Wang, Jooyong Yi, Xiao Liang Yu, Pinghai Yuan and his daughter, Jiang Zhang, Jingfeng Zhang, Yuntong Zhang, etc. Because of all of them, I have a such wonderful journey at NUS and Singapore.

Last but not least, I would like to thank my family, my parents for raising me up to more than I can be. They taught me everything via actions instead of words, they always encourage me to do what I like and always support me. I would also like to express my biggest thank to Cassie Cheng for her love, her patience and understanding, her support and encouragement, and for putting up with many troubles that are due to me following the academic path. Her unconditional love and support are always my greatest impetus.

(All the names are listed in alphabetical order).

Contents

Abstract

Programming-by-example (PbE) is one of the well-studied auto-programming techniques. PbE systems attempt to inductively construct programs according to the specification demonstrated using concrete examples. However, the examples, which are usually represented in a form of input-output pairs, can only specify part of the behaviors of the expected program. If the given examples are incomplete, which is common in practice, the auto-generated program can easily overfit the given examples, i.e. the auto-generated program shows correct behavior on the given examples, but cannot be generalized to the inputs outside the given examples.

Early test-driven program repair and program synthesis can be seen as two instances of PbE systems. Test-driven program repair seeks to rectify program bugs by automatically generating patches. Repair techniques are driven by a correctness criterion which is in the form of a test suite. Such test-based repair may produce overfitting patches, i.e., the patched programs show correct behavior on the given tests, but still fail on tests outside the test-suite driving the repair. Similarly, program synthesis, the technique that automatically generates programs according to given input-output examples, also prone to synthesize overfitting programs. In the literature of program repair and program synthesis, this is called *overfitting* problem.

This work introduces a series of approaches to advance PbE techniques by alleviating the overfitting issues in program repair and synthesis. Our approaches are united by the idea of using program analysis to strengthen the specifications demonstrated via input-output examples. First, since the tests for repair systems are usually incomplete, we propose an approach to intelligently generate more tests to strengthen the given tests. Second, even though test generation can help discard overfitted patches, it does not provide formal guarantees. To solve this problem, we present a repair method that completely fixes program vulnerabilities via semantic reasoning. Third, inspired by semi-supervised learning, we propose semi-supervised synthesis. Instead of only relying on input-output examples, semi-supervised synthesis also considers the additional inputs, where input-output examples correspond to labeled data and additional inputs correspond to unlabelled data. Last, we propose

an output-oriented synthesis to synthesize programs relying on both input-output examples and additional outputs.

Our experiments showed that the proposed techniques advance the state of the art of program repair/synthesis by discarding overfitted patches/programs and increasing the quality of automatically generated patches/programs. We think this is an important step forward to apply program repair in the real world and help end-users via program synthesis.

List of Figures

List of Tables

Publications Appeared

[1] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair", in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Beijing, China: ACM, 2019, pp. 8–18.

[2] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2020.

[3] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, and A. Tiwari, "Feedback-driven semi-supervised synthesis of program transformations", *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[4] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks", in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1147–1158.

[5] X. Gao and A. Roychoudhury, "Interactive patch generation and suggestion", in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 17–18.

[6] X. Gao, A. Radhakrishna, G. Soares, R. Shariffdeen, S. Gulwani, and A. Roychoudhury, "Apifix: Output-oriented program synthesis for combating breaking changes in libraries", *Proc. ACM Program. Lang.*, no. OOPSLA, 2021.

*Note that, the research presented at paper [3] and [4] was in collaboration with Fujitsu Lab and Microsoft, as part of my internship.

Chapter 1

Introduction

The world is increasingly dependent on software, from large programs in the cloud to cell phone applications in our pocket. As every developer knows, developing and maintaining software are difficult and extremely time-consuming. Researchers have proposed auto-programming techniques [8] to help developers fix software bugs [65, 64, 93] (write patches) or even write programs [48, 109, 1]. However, it is very hard to ensure automatically generated patches or programs reflect the intent of developers. Automatically generating high-quality programs is crucial to release developers from burdensome development and maintenance tasks.

Programming-by-example (PbE) [39] is one of the well-studied techniques that can automatically generate programs. PbE systems construct programs according to specifications demonstrated via concrete examples. It inductively infers generalized programs that can be used on new inputs. *Test-driven program repair* and *program synthesis* can be seen as two instances of PbE systems. Automated program repair [65] is an emerging area for automated rectification of programming errors. In the most commonly studied problem formulation, given buggy program P and a set of passing tests pT and failing test-suite fT, the goal of repair is to find a (minimal) change to P and make it pass all tests from pT and fT. Instead of generating patches, program synthesis studies the problem of constructing the whole or partial programs according to given input-output examples. For a given input domain $\mathbb{I}$ and output domain $\mathbb{O}$, program synthesis systems take as input a set of examples $\{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_n \mapsto \mathsf{o}_n\}$ and produce a program $\mathsf{P} : \mathbb{I} \to \mathbb{O}$ such that $\mathsf{P}(\mathsf{i}_k) = \mathsf{o}_k$ for all $0 \leq k \leq n$.

However, in practice, the given test suite or examples are usually incomplete

program specifications. Driven by the incomplete specifications, the fixed program may pass all the given test-suite, but may still be buggy on the inputs outside pT and fT. The problem is particularly dangerous when fixing software vulnerabilities. If the correctness specification driving the repair is incomplete, the patched program may be still vulnerable. Specifically, Smith, et al [121] show that GenProg [64], one of the well-studied APR tools, do not generate patches that substantially improve test suite performance on the held-out set. TrpAutoRepair [102] is likely to break undertested functionality when patching programs, such that the "patched" program is even worse than the un-patched program. For automatically generated fixes of program bugs, we need a stronger level of assurance about the quality of patches. Similarly, the synthesized programs work on the given input-output examples $\{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_n \mapsto \mathsf{o}_n\}$, but may not be generalizable to inputs outside the given examples. The overfitted programs may produce false positives or false negatives, which will lose the trust of users [33]. In the literature of program repair and program synthesis, this is called *overfitting* problem [121, 104, 60]. Overfitting is one of the most challenging problems that prevent PbE systems from generating high-quality patches or programs.

Existing approaches have tried to handle the overfitting problem in different ways. Since the specifications demonstrated via test cases are usually incomplete, automatically generating more tests is a direct and useful strategy to alleviate the overfitting problem in program repair. Existing approaches generate additional test cases using symbolic execution, grey box fuzzing [146] (e.g AFL) or evolutionary algorithm [147] (e.g. EvoSuite [30]). However, we argue those approaches are inefficient in generating useful tests for repair systems. This is because existing test generation techniques are designed for general testing purposes and they do not consider the semantic of program patches. The second approach to address overfitting is heuristically ranking candidate patches or programs. Typical approaches rank patches or programs according to statistical information learned from code repositories [80, 70, 62, 113] or predefined heuristics [109]. However, there is no guarantee that patches or programs are ranked in the order of likelihood to be correct. Another way to address overfitting is by using reference implementation. In many development scenarios, there exists a reference implementation [77]. The automatically generated program should not only pass the given tests/examples, but

also be compatible with the reference implementation. Unfortunately, the reference implementations are not always available.

To handle the above limitations, the goals of this thesis are to improve PbE systems and alleviate the overfitting issues both in program repair and program synthesis. Specifically, we propose a series of techniques that consist of the following tightly connected components:

- **An approach that alleviates overfitting via intelligent test generation** To efficiently filter out overfitted patches, we propose to tightly integrate testing and program repair. Specifically, our approach fuses test and patch generation into a single process, in which patches are generated with the goal of passing existing tests, and new tests are generated with the objective of filtering out overfitted patches by distinguishing candidate patches in terms of behavior. We use crash-freedom as the oracle to discard patch candidates which crash on the new tests. At its core, our approach defines a grey-box fuzzing strategy that gives higher priority to new tests that separate patches behaving equivalently on existing tests. This test generation strategy identifies semantic differences between patch candidates and reduces overfitting in program repair.

- **An approach that alleviates overfitting via symbolic reasoning** Even though test generation can help discard overfitted patches, it does not guarantee that the generated patches completely fix a bug. To solve this problem, we present a new mechanism for automatically generating patches to completely repair security vulnerabilities. Given a program vulnerability witnessed by some crashing input or exploit, our basic approach relies on extracting a constraint representation of the underlying cause of the crash. This crash constraint can then be propagated backward to guide program repair over the space of possible fix locations. Our method synthesizes patches that are guaranteed to avoid repeating the original crash, thereby repairing the program for all program inputs not limited to a specific test suite. As such, our approach helps address the test overfitting problem.

- **An approach that alleviates overfitting via semi-supervised synthesis** Existing program synthesis takes input-output examples as correctness specifica-

tion, which may lead to overfitted programs. Inspire by semi-supervised learning which learns models from both labeled data and unlabelled data, we propose semi-supervised synthesis. We take a novel view of the overfitting problem as a semi-supervised synthesis problem: apart from the concrete input-output examples which correspond to labeled data, the synthesis procedure also exploits access to additional inputs which correspond to unlabelled data. The main insight is that the additional inputs are the inputs that should be incorporated into the input domain of the synthesized programs. We present a procedure to solve the semi-supervised synthesis problem using anti-unification and existing programming-by-example synthesis technology. Specifically, our technique first infers the corresponding outputs for the additional inputs to produce a set of additional examples. Then, we use existing synthesis engine to synthesize programs according to given examples and the inferred additional examples. To eliminate reliance on access to marked additional inputs, we generalize the semi-supervised synthesis procedure to a feedback-driven procedure that also generates the marked additional inputs in an iterative loop.

- **An approach that alleviates overfitting via output-oriented synthesis** Apart from additional inputs, we also observe that there are a large number of available additional outputs. The additional outputs are a set of outputs without corresponding inputs being available. We explore the research question: *whether the additional outputs can be used to synthesize programs.* Our main insight is that the additional outputs are embedded with human intelligence, which demonstrates the domain and structure of the expected outputs. We take a novel view of this problem as output-oriented program synthesis, which synthesizes programs based on both input-output examples and additional outputs. We solve the output-oriented program synthesis problem using an approach that is similar to semi-supervised synthesis. Specifically, our technique infers the corresponding inputs for the additional outputs to produce additional examples, which are used to synthesize programs. We show an application of using output-oriented program synthesis in automated API usage adaptations.

Tools and evaluation. Based on these proposed approaches, we have developed four tools, Fix2Fit, ExtractFix, BluePencil$_{\mathsf{auto}}$, and APIfix, respectively.

Fix2Fit and ExtractFix are then evaluated on a set of real-world vulnerabilities (CVEs and a set of bugs detected by Google OSS-Fuzz). By generating more test cases, Fix2Fit rules out a large part (>60%) of overfitted patches that partially fix the bug or introduce new bugs. While Fix2Fit does *not* provide formal guarantees about crash/vulnerability-freedom, cross-validation with fuzzing tools and their sanitizers provides greater confidence about the produced patches. Meanwhile, with the guidance of constraints, ExtractFix shows its efficacy in fixing a wide range of vulnerabilities. Specifically, among 30 bugs, it successfully produces 16 patches that are semantically equivalent to developer patches, outperforming all the existing patch generation techniques. Further, we also demonstrate that the presented techniques can significantly reduce the overfitting problem in program synthesis by showing two applications on program transformation. First, we applied semi-supervised synthesis ideas to learn program transformations for automating repetitive edits. Compared to existing tools, our semi-supervised approach is vastly more effective in synthesizing correct programs with significantly lesser amounts of examples. It improves the recall of generating correct edit suggestions from 27% to 100% while keeping high precision. Second, in the scenario of automating API usage adaptions, our technique learns adaptation rules according to human adaptations and the usages of the new library version, and automatically transforms the API usages that are still relying on the old library version. We show that our tool achieves around 90% accuracy in correctly adapting API usages, which is much higher than state-of-the-art tools.

The proposed approaches impact the current state of practice by improving the quality of auto-generated patches/programs. First, the developed system enables developers to automatically and efficiently repair real-world defects and vulnerabilities, such as CVEs and the bugs detected by the Google OSS-Fuzz infrastructure. Specifically, our main contribution is to provide more guarantees to completely fix bugs and increase confidence about the correctness of the auto-generated patches. Second, the developed system based on program synthesis helps developers to automatically and efficiently maintain software systems, such as code refactoring [83] and API usage adaptation [92]. Specifically, our approach could synthesize high-quality transformation rules with fewer human-provided examples. The synthesized transformation rules can then automate repetitive tasks in the software maintenance process. One of the developed tools is deploying into the Microsoft Visual Studio.

In summary, our approach can impact developers' productivity and increase the quality of software.

Research Scope. In this thesis, we restrict our investigation to fix security-related crashes and vulnerabilities (e.g., CVEs). Fixing crashes and vulnerabilities are usually more critical than fixing normal bugs, since the delay in fixing security-related bugs will expose software systems to malicious attacks. Although the presented approaches are investigated in the context of fixing crashes and vulnerabilities, they have the potential to be used to fix other kinds of bugs. To fix crashes and vulnerabilities, the proposed approaches are not guaranteeing to generate correct patches. Instead, they are designed to decrease the reliance on the test cases and alleviate the overfitting problem by completely fixing a vulnerability, i.e., not only fixing the given failing tests, but also fixing all the inputs that can trigger this vulnerability. Furthermore, we just consider the candidate patches that only modify program expressions, although, our approaches can be extended to support other forms of patches. Besides fixing bugs, we also apply the presented techniques to automate program transformations, including automating repetitive edits and automated API adaptation. Although those two applications are in different contexts, similar to program repair, we also focus on the overfitting problem in synthesizing transformation rules according to examples/tests. Beyond those two use cases, we believe the proposed approaches can also be used on other software maintenance and development tasks.

Organization. The remainder of this thesis is organized as follows. We first discuss prerequisite knowledge and background in Chapter 2. In Chapter 3 and 4, we present our approaches that alleviate the overfitting problem of program repair via intelligent test generation and semantic reasoning, respectively. Chapter 5 presents the semi-supervised program synthesis, and Chapter 6 introduces our output-oriented program synthesis and its application in API usage adaptations. Chapter 7 presents the related work and Chapter 8 concludes this thesis as well as discuss potential future research directions.

Chapter 2

Background

In this chapter, we recap the essential background knowledge including: program repair, program synthesis, and grey-box fuzzing.

2.1 Program Repair

We denote a program as p and a patched program obtained from p by substituting an expression e with e' as $p[e \mapsto e']$. The substitution $(e \mapsto e')$ of expressions is called *patch* of p, and sets of *patches* are denoted as $\{p_0, p_1, ..., p_n\}$. $T = \{t, t_1, ..., t_m\}$ represent a set of program inputs (test suites). Automated program repair techniques take in a buggy program, and a set of passing and failing tests, and aim to generate a patched program that passes all the given tests. Several automatic program repair techniques have been proposed to generate patches for buggy programs.

2.1.1 Search-Based Repair

Search-based repair systems, e.g. GenProg [64], SPR [68] and Prophet [70], first generate a patch space S [69] and then search among S to find patches that can pass all the given tests. Typical search-based repair generates patch candidate space S using a set of predefined repair operators. The operators defines how to mutate or change the buggy program to generate new program variants as repair candidates. Among the patch space, it then use a search algorithm to find correct patches based on some search heuristics. This process terminates when any program patch passes all given test cases is found or when all patches have been evaluated. Patches that pass all the given tests are called *plausible* patches.

This traditional algorithm enumerates and tests patches one by one, which is not efficient. Therefore, it scales only to small search spaces because of the cost of

test execution. Test-equivalence analysis [52, 58, 75] can optimize this process.

Definition 2.1.1 (Test-equivalence) *Let p and p' be programs, t be a test. We say that p is test-equivalent to p' w.r.t. t if both p and p' produce same output by executing t.*

In some cases, the test-equivalence of two programs can be detected without executing each of them individually, but instead performing dynamic analysis while executing only one of them, which helps to reduce the number of test executions required for evaluation. In this work, we consider one such analysis referred to as *value-based test-equivalence* [75]. The search space of patches is represented as a collection of patch partitions. The patch partitions are constructed by using a value-based test-equivalence relation.

Definition 2.1.2 (Value-based test-equivalence) *Let e and e' be expressions, p and p' be programs such that $p' = p[e \mapsto e']$, t be a test. We say that p is value-based test-equivalent to p' w.r.t. t if e is evaluated into the same sequence of values during the execution of p with t, as e' during the execution of p' with t.*

In this thesis, we consider the search spaces of candidate patches that consist of only modifications of program expressions. The search space is defined by the following transformation schemas:

- Change an existing assignment:

$$\texttt{x} \coloneqq \texttt{e}; \quad \mapsto \quad \texttt{x} \coloneqq \texttt{e}';$$

- Change an existing if-condition:

$$\texttt{if (e) \{...\}} \quad \mapsto \quad \texttt{if (e') \{...\}}$$

- Add an if-guard to an existing statement `S`:

$$\texttt{S}; \quad \mapsto \quad \texttt{if (e) S};$$

where e and e' are arbitrary expressions of bounded size.

2.1.2 Semantic-Based Repair

Different from search-based repair, the semantic-based program repair techniques first construct constraints that should be satisfied to fix the defect, and then they use program synthesis to directly synthesize patches that satisfy the repair constraints. Typical semantic-based repair, e.g., SemFix [93], Nopol [144], SPR [68] and Angelix [79], replaces the suspicious expression by a symbolic expression x. It then executes each test from the beginning. When the execution reaches the suspicious location, it then starts to execute the program symbolically with x as the symbolic variable. This execution yields a set of formulas on x that can be used as a specification of the program semantics. According to the given test oracle and the formulas on x, it constructs a set of constraints on x that can satisfy the test oracle (pass tests). Program synthesis techniques, e.g., component-based synthesis [48], are utilized to find an expression that satisfies the inferred constraints obtained from the previous step. Replacing the suspicious expression with the synthesized expression ensures that all the tests can be passed. Similar to search-based approaches, semantic-based approaches also suffer from the overfitting issue since they purely rely on test cases as specifications.

2.1.3 Learning-Based Repair

Another line of repair is to learn repair strategies from human patches, such as Genesis [67], GetaFix [7] and Phoenix [9]. Those techniques take the patch generation as a code transformation problem. First, they mine human patches that fix defects in existing software repositories. They then represent the repair operation as two template abstract syntax trees (ASTs) $\tau_i \mapsto \tau_o$. One template AST τ_i matches the code in the original program. The other template AST τ_o specifies the replacement code for the generated patch. The abstract template ASTs $\tau_i \mapsto \tau_o$ is then applied to the buggy programs to produce patches. The repairability of those techniques does not rely on predefined transformation operators. Instead, they can automatically learn repair strategies from available human patches.

2.2 Program Synthesis

Given a set of specifications, program synthesis generates a program satisfying the specifications. Specifically, for a given input domain $\mathbb{I}$ and output domain $\mathbb{O}$, a program synthesis technique takes as input a set of examples $\{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_n \mapsto \mathsf{o}_n\}$ and produces a program $\mathsf{P} : \mathbb{I} \to \mathbb{O}$ such that $\mathsf{P}(\mathsf{i}_k) = \mathsf{o}_k$ for all $0 \leq k \leq n$.

2.2.1 Program Synthesis as Second-Order Constraint Solving

Program synthesis is formalized to be a second-order constraint solving problem in the recent work on SE-ESOC [76]. We build our program synthesizer on top of the approach proposed by SE-ESOC. Given a set of components C, this approach first constructs a set of terms and then represents them as a tree. Specifically, each leaf of the tree corresponds to components without input, and an intermediate node has as many subnodes as the maximal number of inputs of a component. Figure 2.1 shows a tree with three nodes, and each node is constructed using four components ("x", "y", "+", "-"). The leaf nodes 2 and 3 do not have subnodes, while node 1 has two subnodes since components "+" and "-" takes two inputs. For each node i with sub-node $\{i_1,i_2,...,i_k\}$, its output is represented as out_i, and its inputs is represented by $\{out_{i_1},out_{i_2},...,out_{i_k}\}$ (the output of subnodes). In addition, boolean variables s_i^j is the j-th selector of node i, which means j-th component is used in this node, F_j represents the semantics of j-th component, and N is the number of nodes in the tree. For the tree in Figure 2.1, with $\{s_1^3, s_2^1, s_3^2\}$ as true, the output of the root node will be $x + y$. The well-formedness constraint is encoded as $\varphi_{wfp} := \varphi_{node} \wedge \varphi_{choice}$, such that:

$$\varphi_{node} := \bigwedge_{i=1}^{N} \bigwedge_{j=1}^{|C|} \left(s_i^j \Rightarrow \left(out_i = F_j \left(out_{i_1}, out_{i_2}, ..., out_{i_k} \right) \right) \right) \tag{2.1}$$

$$\varphi_{choice} := \bigwedge_{i=1}^{N} exactlyOne \left(s_i^1, s_i^2, ..., s_i^C \right) \tag{2.2}$$

For a node, φ_{node} describes the semantic relations of each node between its output and inputs, where the inputs are the outputs of its sub-nodes. φ_{choice} restricts that only exactly one component is selected inside each node. Using the above encoding,

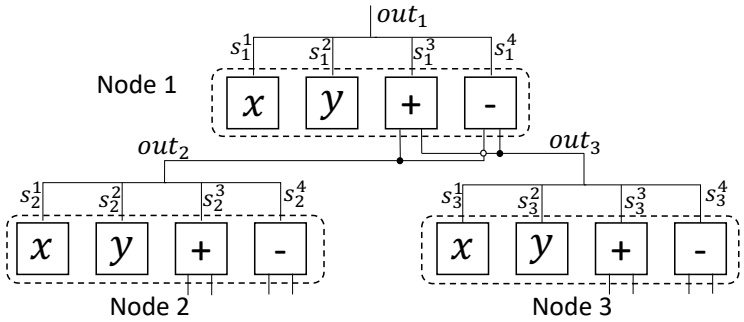


Figure 2.1: SE-ESOC encoding with four components ("x", "y", "+", "-") and three nodes.

```
program     := (guard, transformer)
guard       := pred | Conjunction(pred, guard)
pred        := IsNthChild(node, n) | IsKind(node, kind)
               | Attribute(node, attr) = value | Not(pred)
node        := Path(input, path)
transformer := construct | select
construct   := Tree(kind, attrs, children)
children    := EmptyChildren | Cons(node, children)
               | InsertChild(Children(select), pos, node)
               | DeleteChild(Children(select), pos)
               | ReplaceChildren(Children(select), posList, children)
               | MapChildren(λ input: transformer, Children(select))
select      := Nth(Filter(guard, SubTrees(input)), n)
pos         := n | ChildIndexOf(node)

Variables:
  AST input;  List<int> posList;   string kind, attr, value;
  int n;      XPath path;          Dictionary<string, string> attrs;
```

Figure 2.2: Domain-specific language for edit programs

the output of the root node represents a function f that connects inputs and outputs of components. Finally, given n input-output pairs $\{(\alpha_k, \beta_k) \mid 1 \leq k \leq n\}$, the synthesis goal is to generate function f by traversing the abstract tree and satisfying $\varphi_{correct}$, where

$$\varphi_{correct} := \bigwedge_{k=1}^{n} \beta_k = f(\alpha_k) \tag{2.3}$$

2.2.2 Program Synthesis for Code Transformation

In chapter 5, we apply program synthesis for code transformations. Here, we describe how program synthesis techniques can be used to automatically transform codes. In the domain of code transformation, both input $\mathbb{I}$ and output $\mathbb{O}$ are fixed as Abstract Syntax Tree (AST). ReFazer [109] is one of the well-known program synthesis engine for code transformations. In ReFazer, the programs are drawn from the domain-specific language (DSL) shown in Figure 2.2. In this DSL, a transformation program is formulated as a function pair (`guard`, `transformer`).

- `guard` : $\mathbb{T} \rightarrow$ `Boolean` The guard defines the context where the transformation rule is applied. The `guard` is composed of a set of conjunctive predicates on the attributes (e.g., Type, Kind, TextValue, etc) of the AST and its sub-trees. The `guard` evaluates where an AST satisfy its predicate and return a `Boolean` value accordingly.

- `transformer` : $\mathbb{T} \rightarrow \mathbb{T}$ The transformation defines how to transform an input AST to an output AST. In our setting, `transformer` recursively constructs the output AST using two operators: (1) `select`: A `select` returns a subtree of the input. The subtree is identified as the n^{th} node that satisfies a guard, and (2) `construct`: A `construct` returns a subtree that is built by specifying the kind of node, its attributes and children. The children may be constructed using several different operators. For example, operator `InsertChild(select, pos, node)` selects a node (called parent) from the input and returns the parent's children with an additional node at the position `pos`.

Essentially, the `guard` determines which AST should be transformed, while the `transformer` specifies how the AST should be transformed. Formally, we have that $P(T) =$ `transformer`(T) if `guard`$(T) =$ `true`, and $P(T) = \perp$ otherwise. In general, given a set of input-output examples $\{i_0 \mapsto o_0, \ldots, i_n \mapsto o_n\}$, synthesizing a transformation program P:(`guard`, `transformer`) is a generalization process of the concrete transformation examples, such that `guard`$(i_k) =$ `true` and `transformer`(i_k) $= o_k$ for all $k \in 0 \ldots n$.

We do not provide details on how ReFazer synthesizes programs given examples. However, one important aspect of the ReFazer synthesis algorithm is that it prefers

selections over constructions, i.e., when a particular subtree of the output can be selected from the input AST, ReFazer returns a program with the selection. The reader is referred to [109, 100] for further details.

Over-generalization and Under-generalization. Input-output examples are inherently an under-specification of the intended program, and any program synthesis technique needs to generalize inductively from the examples. Developers view false positives more unfavorably than false negatives—it causes them to lose trust in the tool [10]. Hence, many synthesis techniques, including ReFazer, used in the source code transformation domain err on the side of under-generalization (for examples, see [7, 82, 109]).

2.3 Greybox Fuzzing

Algorithm 1: Greybox Fuzzing

```
   Input: seed inputs T
1  while timeout is not reached do
2  |  t := chooseNext(T);
3  |  energy := assignEnergy(t);
4  |  for i from 1 to energy do
5  |  |  t' := mutate(t);
6  |  |  if isInteresting(t') then
7  |  |  |  T := T ∪ t';
8  |  end
9  end
```

Generating more test cases for repair system is one way to alleviate its overfitting problem. We briefly describe how Greybox Fuzzing (e.g. AFL [132]) generates test cases in Algorithm 1. Given a set of initial seed inputs T, the fuzzer chooses t from T (line 2) in a continuous loop. For each selected t, the fuzzer determines the number of tests to be generated by mutating t, which is called the *energy* of t, and its assignment is dictated by a *power schedule*. The fuzzer generates new inputs by mutating t according to defined mutation operators and the power schedule. New input t' will be added to the circular seed queue (line 7) for further mutation if it is a "interesting" input, meaning it potentially exposes new control flows as deemed from the compile-time instrumentation.

Directed Grey-Box Fuzzing. AFLGo [11], an extension of the popular grey-box fuzzer AFL, directs the search to given target locations. In AFLGo, an estimation of the distance of any basic block to the target(s) is instrumented at compile time, and these estimates are used during test generation to direct the search to the targets. Specifically, tests with lower estimated distance to the target are preferred by assigning more energy to these tests, and this energy difference increases as temperature decreases. The temperature is controlled by a *cooling schedule* [55], which dictates how the temperature decreases over time. Based on *cooling schedule*, the current temperature T_{exp} is defined as:

$$T_{exp} = 20^{-\frac{ctime}{time_x}} \tag{2.4}$$

where $time_x$ is user-defined time to enter "exploitation" (preferring tests deemed closer to the target) from exploration, *ctime* is current execution time. Given the current temperature T_{exp}, normalized distance $d(t, T_b)$ between test t and target location T_b, AFLGo introduces an annealing-based power schedule (APS):

$$aps(t) = (1 - d(t, T_b)) * (1 - T_{exp}) + 0.5T_{exp} \tag{2.5}$$

and determines the energy assigned to t by multiplying the energy assigned by AFL with a power factor calculated using APS:

$$energy_{aflgo}(t) = energy_{afl}(t) * 2^{10*aps(t)-5} \tag{2.6}$$

Chapter 3

Alleviate Overfitting via Intelligent Test Generation

To alleviate Overfitting problem in program repair, this chapter present an intelligent test case generation that can help discard overfitted patches.

3.1 Introduction

Generate-and-validate program repair systems first construct a space of candidate patches (set S in Figure 3.1) and search for a patch that passes the given tests. Such patches that pass given tests are called plausible patches (set P in Figure 3.1). Since a test suite is an incomplete specification, only part of plausible patches may be correct (set C in Figure 3.1), and the remaining patches merely overfit the tests. In rare cases, all of the plausible patches are correct. When we repair a program crash, the overfitted patches may still cause program crashes on the tests outside of the given test suite. In this work, we propose to use test generation to divide the

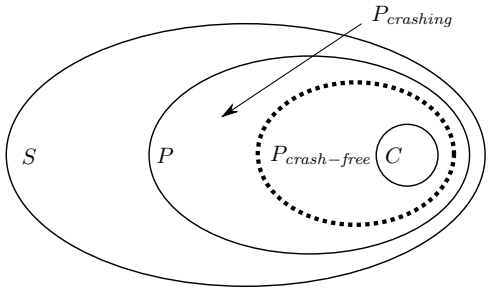


Figure 3.1: Structure of program repair search space, where S is a space of candidate patches, P is a set of plausible patches, $P_{crashfree}$ is a set of crash-free patches, C is a set of correct patches, $P_{crashing} = P \setminus P_{crashfree}$ is a set of crashing patches.

set of plausible patches P into two subsets $P_{crashfree}$ (crash-free plausible patches), and $P_{crashing}$ (crashing plausible patches) and suggest that program repair should aim to find a patch from the set $P_{crashfree}$, that is a patch that passes given tests and does not cause crashes for the inputs outside of the repair test suite. Although crash-freedom is implicitly assumed to hold for correct patches, existing program repair systems do not guarantee this property and may generate patches causing crashes or even introduce new crashes and vulnerabilities.

A prominent group of test case generation techniques that were successfully used to find serious vulnerabilities in popular software is coverage-based greybox fuzzing [132, 137]. These techniques resort to compile-time instrumentation to track coverage information and guide the test generation. In these algorithms, inputs are randomly mutated to generate new inputs, and a higher priority is assigned to inputs that exercise new and interesting program paths. Whether a generated input exercise new paths is predicted based on whether new control flow transitions are exercised. The main intuition of these techniques is that covering more program paths (that correspond to different semantic partitions of the input space) enables them to cover more parts of program functionality and therefore find more crashes.

Coverage-based greybox fuzzing can be applied to detect crashes in automatically generated patches in the following way: (1) generate a high coverage test suite using fuzzing for the original program, and (2) run this test suite on all plausible patches P to discard those that introduce crashes, and thus find an over-approximation of $P_{crashfree}$. However, we argue that this approach is ineffective for the following two reasons. First, each candidate patch alters the semantics of the original program and therefore might induce different semantic partitions of the input space, so tests generated for the original program might not adequately cover the functionality of the patched program. Second, to divide the set of plausible patches P into subsets $P_{crashfree}$ and $P_{crashing}$ (dotted line in Figure 3.1), the generated tests should also *differentiate* patches in the search space.

To take the above considerations into account, we suggest that test generation for program repair should not be based merely on the coverage of the original program, but also on the coverage of the divergences introduced by the patches in the search space. Thus, a test suite produced by our method is not just aimed to cover the functionalities of the original program, but also (1) functionality that is altered by

the candidate patches, and (2) functionality that differs across candidate patches. Since such a test suite is more likely to find divergences among plausible patches P, consequently it is more likely to differentiate between $P_{crashfree}$ and $P_{crashing}$.

As a practical realization of this concept, we propose a new algorithm that fuses patch and test generation into a single process. In this process, patches are generated with the objective of passing existing tests, and new tests are generated with the objective of differentiating patches. However, since there could be many plausible patches, it is inefficient to separately generate tests to distinguish each pair of these patches. Instead, we propose to group patches into test-equivalence classes, sets of patches that demonstrate equivalent behaviour on existing tests. These are called *patch partitions*. When generating tests, we assign higher priority to those tests that refine patch partitions into finer-grained partitions, since such tests cover previously uncovered semantic differences between candidate patches. This allows us to efficiently cover divergences between candidate patches without explicitly considering all pairs of patches. As shown in our evaluation in section 3.5, our approach rules out 18% more overfitted patches than the traditional coverage-based grey-box fuzzing AFL [132].

3.2 Motivating Example

In this section, we give a high-level overview of our approach to generate crash-free patches by presenting an example from *FFmpeg*. *FFmpeg* is a collection of libraries and programs for handling video, audio and other multimedia files, streams. A buffer overflow vulnerability is reported by OSS-Fuzz[1] in May 2017. This vulnerability is caused by incorrect bounds checking when *FFmpeg* decodes DirectDraw Surface (DDS) files[2]. Listing 3.1 shows the key code snippet as well as its patch. The decode method in Listing 3.1 takes four parameters, where *gb* stores the origin data of the input image, *width* and *height* are initialized based on the information from input image header, and *frame* is a buffer to store decoded data. If *remaining_space* is equal to *width+3* (line 15), an invalid buffer access will occur in line 23, since it will overwrite the memory locations after *frame_end*. The correct patch[3] for this

[1]https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=1345
[2]DDS is an image file format for storing texture and environments
[3]https://github.com/FFmpeg/FFmpeg/commit/f52fbf

```
1  int decode_dds1(ByteContext *gb, uint8_t *frame, int width, int
       height){
2   ...
3   segments = bytestream2_get_le16(gb);
4   while(segments--) {
5     if (bitbuf & mask) {
6       ...
7     }
8     else if (bitbuf & (mask << 1)) {
9       v = bytestream2_get_le16(gb)*2;
10      if (frame - frame_end < v)
11        return AVERROR_INVALIDDATA;
12      frame += v;
13    } else {
14      int remaining_space = frame_end-frame;
15      if(remaining_space < width+3)
16      //"width+3" → "width+4" (correct patch)
17        return AVERROR_INVALIDDATA;
18      frame[0] = frame[1] = frame[width] =
19      frame[width+1] = bytestream2_get_byte(gb);
20      frame += 2;
21      frame[0] = frame[1] = frame[width] =
22      //buffer overflow location
23      frame[width+1] = bytestream2_get_byte(gb);
24      frame += 2;
25    }
26  }}
```

Listing 3.1: Buffer overflow vulnerability in *FFmpeg*

vulnerability is to modify the condition in line 15 from *width+3* to *width+4*.

Automated program repair (APR) takes a buggy program and a set of test cases (including failing tests that will cause program crash) as inputs. Since the tests do not cover all program functionalities, APR tools may generate many overfitted patches which make the patched program pass all the test suites but do not actually fix the bug. Given the failing test case and a set of supported transformations, some existing repair tools, e.g., F1X [75], generate a lot of plausible patches (including the correct patch). However, it is hard to select the correct one out of the large number plausible patches. For instance, F1X generates 1807 plausible patches to fix this buffer overflow vulnerability and only two of them are correct. Column *plausible patch* in Table 3.1 shows part of patches that can make the program pass the failing test. Out of them, the fourth and sixth patches are semantically equivalent to the developers' patch. However, other patches overfit the existing test set. Those patches fix the crash triggered by the existing test set, but they do not completely

Table 3.1: Plausible patches and their behaviors on new test

Id	plausible patch	T_1	T_2	T_3	T_4
1	remaining_space>width+1	(T) ✓	(F) ✗	—	—
2	remaining_space>width+2	(F) ✗	—	—	—
3	remaining_space! =width+3	(T) ✓	(T) ✓	(T) ✓	(T) ✓
4	remaining_space<=width+3	(T) ✓	(T) ✓	(F) ✓	(F) ✓
5	remaining_space>=width+3	(F) ✗	—	—	—
6	remaining_space<width+4	(T) ✓	(T) ✓	(F) ✓	(F) ✓
7	remaining_space<width+5	(T) ✓	(T) ✓	(T) ✓	(F) ✓
8	remaining_space<width+6	(T) ✓	(T) ✓	(T) ✓	(F) ✓

T_1: remaining_space=width+2 T_2: remaining_space=width
T_3: remaining_space=width+4 T_4: remaining_space=width+6

fix this vulnerability or even introduce new vulnerabilities (e.g. the patched program using the first patch still crashes on a test that makes *remaining_space* equal to *width+2*). The fundamental reason is that the search space of candidate patches is under-constrained.

To tighten the search space and rule out crashing patches, one solution is to automatically generate more test cases. This leads to the following research question: how to generate test cases that can filter out a large fraction of overfitted patches?

Existing fuzzing techniques are not suitable for efficiently generating tests to constrain the patch space. Most fuzzing tools (e.g. AFL [132]) favor the mutation of input with the goal of finding unexplored statements and enhancing code coverage. Different from program testing, the role that fuzzing plays in repair is to generate test cases to find discrepancies between patches and filter out overfitted patches instead of improving code coverage. In this example, we expect tests that can drive the execution to the patch location with different program states (values of *remaining_space*, *width*).

To efficiently generate test inputs that can filter out overfitted patches and differentiate patches, we propose a strategy to integrate test generation and program repair. Our main intuition is, if one test is able to find the discrepancies between patches, its neighbors are also likely to find discrepancies. Table 3.1 shows the patch behaviors over four tests. The patch behavior is shown by its effectiveness in repairing vulnerability and expression value, where ✓ and ✗ represent whether buffer overflow vulnerability is triggered or not by each test, T and F represent the value of patch

expression (true or false). Suppose these four tests are generated in order, with values of *remaining_space* equals to *width+2, width, width+4*, and *width+6* respectively. For instance, the expression value of patch 2 ($remaining\_space > width + 2$) is false ($F$) under test T_1, and program fixed by this patch still crashes (✗) under T_1, so that patch 2 is filtered out and will not be considered in the following iterations. Test input T_1 is able to find the discrepancies between patches, and rule out two overfitted patches. Correspondingly, T_2 and T_3, which are two neighbors of T_1 (a single increment or decrement mutation over *width* or v on line 9), can also find discrepancies.

To guide the test generation process, Fix2Fit adopts an evolutionary algorithm similar to the popular AFL fuzzer [132]. AFL undergoes compile-time instrumentation to capture control flow edges, and at run-time, it uses the instrumentation to predict whether a newly generated test exposes new control flows. Tests that expose new control flows are favored and they are retained for further examination by mutating them further. In addition to the code coverage based heuristic used in AFL, we propose a new heuristic: we favor tests with greater ability to distinguish plausible patches. In this example, AFL will not retain T_1, since it does not improve code coverage. However, Fix2Fit will retain T_1 for further mutation since T_1 can distinguish patches 2 and 5 from others. Therefore, Fix2Fit has a greater chance of finding tests like T_2 or T_3 via mutation. In addition, the chance of generating tests to find discrepancies across patches can be further increased by assigning higher "energy" to T_1 (meaning more mutations of T_1 will be constructed by the fuzzer).

Out of eight patches given in Table 3.1, three plausible patches (1, 2, 5) can be ruled out, since the program constructed by those patches still crashes over some tests. For the remaining five plausible patches, the patched program does not crash, but the semantic behaviors of them are different (two of them are correct). The remaining incorrect patches cannot be ruled out due to the lack of oracles of the generated tests (expected outputs of the newly generated inputs). If the oracle of certain tests such as T_3 is provided (could come from more fine-grained program analysis or developers), all the incorrect patches can be ruled out.

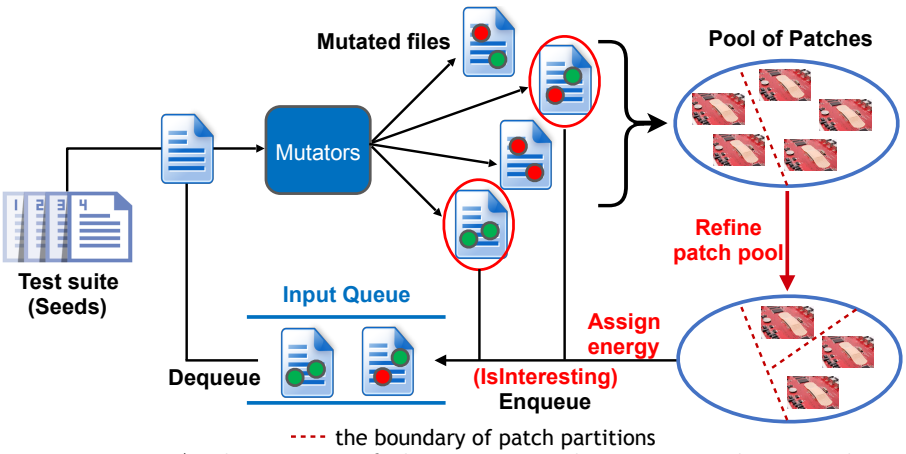


Figure 3.2: Architecture of the integrated testing and repair loop

3.3 Methodology

Fix2Fit is designed to generate new test cases to efficiently rule out overfitted plausible patches and generate crash-free patches. Our goal is to strengthen the filtering of patches by adding additional test cases. Specifically, Fix2Fit observes the semantic differences across plausible patches, and then guides the test generation process. Fix2Fit utilizes the notion of *separability*: the ability of a test to find semantic discrepancies between plausible patches. To represent the semantic discrepancies, we group all patches showing the same semantic behavior under all available test cases into an equivalence class, which is called a patch partition. More formally,

Definition 3.3.1 (Patch Partition) *Let T be a set of available test cases and P be a set of plausible patches of program p. The patched program by patch $p_i \in P$ is denoted as $p[e \mapsto e_i]$. $\forall p_i, p_j \in P$, p_i and p_j belong to same equivalent patch partition if and only if $\forall t \in T$. $p[e \mapsto e_i]$ is value-based test-equivalence to $p[e \mapsto e_j]$ w.r.t t.*

The ability of a test to find semantic discrepancies is formalized as its effectiveness in refining patch partitions. For any two patches p_i, p_j from the same equivalence partition EP, if $p[e \mapsto e_i]$ is not *value-based test-equivalence* (refer to Definition 2.1.2) to $p[e \mapsto e_j]$ w.r.t new test t, we say test t refines partition EP. Different from existing fuzz testing techniques that maximize the code coverage (AFL), or minimize distance to the target location (AFLGo), Fix2Fit is designed to maximize semantic discrepancies across patches (thereby refining patch partitions). To find more

Algorithm 2: Patch-aware Greybox Fuzzing

```
   Input: test suite T, program p
 1 Par := genPlausiblePatches (p, T);          // generate set of patch partitions
 2 pLocs := extractPatchLocs (Par);              // extract set of patch locations
 3 p' := instrument (p, pLocs);                      // instrument fuzzing targets
 4 T_new := { };
 5 while true do
 6 |   t := chooseNext(T);
 7 |   for i from 1 to t.energy do
 8 |   |   t' := mutate(t); T_new := T_new ∪ {t'};
 9 |   |   (isReached, distance, coverage) := exec(p', t');
10 |   |   if isReached then
11 |   |   |   Par := refine_and_filter(Par, t');
   |   |   |   // Break patch partitions & remove overfitted partitions
12 |   |   |   sep := separability(t', T_new) // Equation 3.1
13 |   |   end
14 |   |   t'.energy := powerSchedule(sep, distance, coverage); // Sec 3.3.3
15 |   |   if isInteresting(coverage, sep) then
   |   |   |   // Sec 3.3.4
16 |   |   |   T := T ∪ {t'};
17 |   |   end
18 |   end
19 |   if timeout || sizeOf(Par) == 0 then
20 |   |   break;
21 end
22 Output: remaining patch partitions Par
```

semantic discrepancies between plausible patches, we essentially generate test cases that can make the execution reach the patch location with divergent program states.

3.3.1 Integration of Test Generation and Repair

Figure 3.2 presents the architecture of our integrated testing and repair loop. Given a buggy program and a set of test cases, Fix2Fit first constructs a pool of plausible patches (patch generation). Just like existing grey-box fuzzers, Fix2Fit mutates the given test suite to generate a set of mutations (test generation). Fix2Fit then applies these mutations to the patch candidates to measure their ability to distinguish patches. In this way, the tests which can distinguish existing patches are prioritized and saved as seeds for further mutation. This loop continues with the goal of further breaking patch partitions and ruling out overfitted patches.

Algorithm 2 shows the key steps of Fix2Fit. The main procedure is built on top of an automated patching technique and directed greybox fuzzing technique. Given a buggy program p, a test-suite T, and at least one test case in T that can

trigger a bug, this algorithm will return a set of plausible patch partitions for fixing the bug. FIX2FIT generates the initial set of plausible patches by inheriting the traditional *Generate and Validate* approach, where a set of patch candidates are generated and evaluated using a provided set of test cases (line 1). Incorrect patches are filtered out in the evaluation process, and a set of plausible patches are returned back. Besides plausible patches, it groups patches with the same semantic behavior into a set of patch partitions (as per the value-based test equivalence Definition 3.3.1). The plausible patches may be overfitting, and the patch partitions can be broken by generating more tests.

To filter out overfitted patches by generating new tests, the newly generated tests must at least reach the patch location. We instrument program p with the patch location as target (Line 3) to produce an instrumented program p'. At runtime, the instrumentation is used to calculate code *coverage* and the *distance* to the patch location (line 9), and also the *separability* for each newly generated test. The *separability* of a test t' captures its ability to find semantic discrepancies between plausible patches.

For each newly generated input t', FIX2FIT first evaluates whether t' drives the execution to the patch locations (*isReached*). If test t' reaches any target(Lines 10-12), procedure *refine_and_filter* is invoked, which refines the patch partitions and also filters out patch partitions as follows: (1) *refine_and_filter* refines the current patch partitions *Par* using test t'. The refinement process may break the existing patch partition into several sub-partitions since the underlying value-based test-equivalence relation now also considers the newly generated test t'; (2) After the patch partitions are refined using t', the procedure *refine_and_filter* checks which of the patch partitions can be shown to be overfitting (patches which crash on test t') and filters out those patch partitions.

Separability of a generated test t' (the patch-awareness in our fuzzing method) is exploited along two dimensions: (1) it is used in power schedule to determine the energy assigned to new test t' as shown in line 14, and (2) it is used to determine whether the generated input t' is added to the seed input set T for further investigation/mutation (Lines 15-16).

The integrated fuzzing and repair algorithm is terminated on *timeout*, or when all plausible patches are filtered out.

3.3.2 Separability of Test Cases

In Algorithm 2, test generation is guided by the behavioral differences across plausible patches. The ability of a test to find semantic discrepancies between plausible patches is formalized as *separability*. We now explain how the *separability* is calculated.

When a new test t' is introduced, its effects on the current patch partitions can be captured in two ways: (1) *patch filtering:* rule out crashing patches (2) *partition refinement:* refine existing patch partition into several sub-partitions. Both of these can be used to calculate the *separability* of test t', which in turn determines the "energy" assigned to t' in fuzzing.

We argue that the *partition refinement* is a better heuristic than *patch filtering* for the purpose of guiding fuzzing (see RQ1 of Section 3.5). In the fuzzing process, by mutating a test with high separability, we hope that the generated neighbors are also tests with high separability. If we define separability in terms of the number of overfitted/crashing patches filtered, we note that whether the patch crashes on new test t' or not often depends on very specific values, for instance, a divide-by-zero error can only be triggered when input is 0. Therefore, we cannot assume that by mutating a test that exposes crashes, we are also likely to get tests exposing crashes.

Compared to *patch filtering*, *partition refinement* is a smoother metric, since the patches are grouped into partitions using test-equivalence relation, and whether partitions can be refined only depends on the values of patch expressions. In other words, if one test t' is able to pin-point semantic differences between patch candidates (refine patch partitions), its neighbors (obtained by mutating t') also have a high chance to find semantic differences between patch candidates. Once we generate one test that can refine patch partitions, it is more likely that we can distinguish the crash-free patches from crashing patches, and as a result, rule out overfitted patches. Based on this intuition, we define the *separability* of a test as its ability to refine test-equivalence based patch partitions.

Our notion of *separability* judges how much refinement is observed on the patch partitions once a new test is introduced. Given a set of patch partitions $\{P_1, P_2, ...P_n\}$, and a newly generated test t', if the patches in partition P_i show different behaviors on test t', we say t' refines partition P_i. We use $b(t')$ to represent the number of

patch partitions that can be refined by test t'. FIX2FIT always maintains a set T_{new} of newly generated tests, as shown in Algorithm 2. We define the *separability* of test t' as $b(t')$ divided by maximum $b(t)$ of any pre-generated test $t \in T_{new}$:

$$separability(t') = \frac{b(t')}{\max_{t \in T_{new}} b(t)} \tag{3.1}$$

3.3.3 Power Schedule

We now define the notion of power schedule, which is a measure of the "energy". The energy represents the number of neighborhoods of a test that are investigated (line 14 of Algorithm 2). Our goal is to prioritize the test that can differentiate between plausible patch candidates by mutating it more times and generating more neighborhoods.

To differentiate plausible patches in the search space, we should first generate tests that reach patch location. Therefore, we inherit the power schedule of the directed grey-box fuzzer AFLGo [11], which directs the search to given target locations. Specifically, tests with a lower estimated distance to the target are preferred by assigning more energy to these tests. Apart from reaching patch locations, generating different program states in the patch location is necessary to differentiate plausible patches. FIX2FIT prioritizes the tests with higher *separability* by assigning more energy to these tests.

To generate different program states at the patch location, two kinds of tests are needed: (1) tests that make execution reach patch location following various paths (2) tests that make execution reach patch location following the same path but with different values (to refine value-based test-equivalence relation). To take both kinds of tests into consideration, we utilize the *cooling schedule* [55] notion adapted from simulated annealing. In the cooling schedule, the degree to which a test with high *separability* is preferred over a test with low *separability*. The energy of a test with non-zero *separability* is increased over execution time, i.e., "temperature decreases" using the simulated annealing terminology. In other words, FIX2FIT performs exploration at the very beginning to explore various paths, and gradually changes to exploitation to differentiate plausible patches. Given current temperature

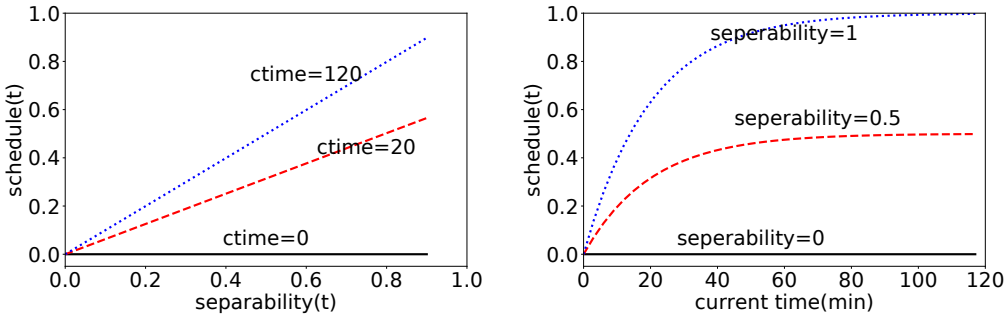


Figure 3.3: (a) energy of a test with different separability at 0min, 20min, 120min (b) energy of a test t at different time when separability(t)=0, 0.5, 1. $time_x$=60min

T_{exp} (as defined in Equation 2.4) as well the *separability(t')*, our power schedule is:

$$schedule(t') = separability(t') * (1 - T_{exp}) \tag{3.2}$$

Thus $schedule(t') \in [0, 1]$. The high-level behavior of this power schedule is illustrated in Figure 3.3. We describe the integration of this power schedule into a fuzzer. Suppose $energy_{aflgo}(t')$ is the energy assigned to t' by AFLGo, we define the integrated energy as:

$$energy(t') = energy_{aflgo}(t') * 2^{schedule(t') * log_2 Max\_Factor} \tag{3.3}$$

where Max_Factor is the user-defined max factor integrated to existing energy, and $\frac{energy(t')}{energy_{aflgo}(t')} \in [1, Max\_Factor]$.

3.3.4 Is Interesting?

Coverage-based greybox fuzzers always maintain a seed queue to save "interesting" tests for further mutation and investigation. This appears as the procedure *isInteresting* in line 15 of Algorithm 2. In existing coverage-based grey-box fuzzers, a test is deemed "interesting", if it is predicted to expose new control flows (and hence improve code coverage); the prediction about discovering new control flows is aided by compile-time instrumentation. Apart from tests that improve code coverage, Fix2Fit also regards the tests with non-zero *separability* as "interesting" and adds them to the seed queue for further mutation. As a result, we retain tests that are capable of distinguishing between existing patch partitions, and the mutations of such tests are examined by the fuzzer in Algorithm 2.

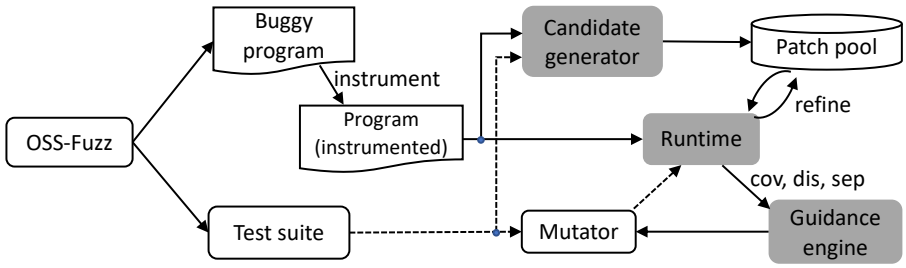


Figure 3.4: Architecture of tool Fix2Fit

3.3.5 Sanitizer as Oracles

The absence of program crashes may not be sufficient to guarantee program correctness. To mitigate this problem, we enhance patch checking by introducing sanitizers. Sanitizers can detect various vulnerabilities at run-time with the help of compile-time instrumentation. Generally, sanitizers convert the software vulnerabilities into normal crashes, e.g. AddressSanitizer crashes the program if a buffer overflow is detected. By using sanitizers we can rule out the patches that introduce vulnerabilities. As compared to only filtering patches based on crashes, more patches can be filtered out.

The sanitizers used by Fix2Fit include UndefinedBehaviorSanitizer[4] (UBSan) and AddressSanitizer[5] (ASan). UBSan is used to catch various kinds of undefined behaviors during program execution, *e.g.* using a misaligned or null pointer, signed integer overflow. ASan is a tool that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer. The patch partitions are not only checked for crashes but are also checked against all available sanitizers, so that remaining patches are guaranteed not to introduce security vulnerabilities in terms of all available tests.

3.4 Implementation

The architecture of Fix2Fit is shown in Figure 3.4. Fix2Fit takes as inputs the buggy program and test suites extracted from OSS-Fuzz benchmark, and generates a set of crash-free patches. The initial test-suite is composed of available developer test cases and the failing tests generated OSS-fuzz. Fix2Fit consists of three

[4]UBSan website: `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
[5]ASan website: `https://clang.llvm.org/docs/AddressSanitizer.html`

main components: *Candidate generator*, *Runtime* and *Guidance engine*. *Candidate Generator* takes the buggy program and tests as inputs and generates a pool of patch candidates. The *Runtime* executes test on the instrumented program and collects necessary information (e.g. code coverage and separability). Accordingly, the *Patch pool* is refined after executing each test. *Guidance engine* is used to guide the fuzzing according to all the information collected at runtime.

Instrumentation: To enable Fix2Fit's grey-box guidance, we first of all instrument the buggy program to gather run-time information. To collect the distance to patch locations, we inherit the instrumentation strategy used in AFLGo [11], where the estimated distances between basic blocks are calculated and injected at compile-time. Besides, we insert a logging instruction after each basic block to collect the execution trace, which is then used for fault localization and for determining whether the patch location is reached. To enhance the checking of patch candidates, we instrument the buggy program using Clang's sanitizers, including Undefined Behavior Sanitizer (UBSan) and Address Sanitizer (ASan). After the instrumentation with sanitizers, we can treat the violation of sanitizer as normal program crash.

Candidate Generator We first generate the search space according to pre-defined transformation operators. The transformations supported in our prototype include: changing the right-hand side of an assignment, condition refinement and adding if-guard. All the operators are borrowed from Prophet [70], Angelix [79] or F1X [75]. The plausible patch candidates are grouped into patch partitions based on their runtime value. To collect the run-time values of patches, Fix2Fit synthesizes a procedure, say $proc_{allpatch}$ enumerating all plausible patches, and generates a meta-program by dynamically replacing the to-be-fixed expression with a call to this procedure. At runtime, the procedure $proc_{allpatch}$ is invoked when the patch location is reached. By controlling the enumeration strategy, this procedure $proc_{allpatch}$ can generate run-time values for all the patches with one run and can select the run-time value of one particular patch to return. This mechanism enables us to generate and refine patch partitions with one run for each test. Patch partitions are maintained in the *patch pool*, as in the F1X repair tool [75]; different from F1X, patch partitions are used to guide test generation with the objective of ruling out patches.

Runtime* and *Guidance engine The main procedure of fuzzing is built on

Table 3.2: Subject programs

Subject	#Defect	#Test	Description
Proj.4	10	3	cartographic projection and geodetic transformation library
FFmpeg	26	11	audio & video processing library
Libarchive	12	4	multi-format archive library
Openjpeg	12	13	open-source library to encode and decode JPEG 2000 images
Libssh	8	23	C library for the SSHv2 protocol
Libchewing	13	11	phonetic input method library
Total	81	—	—

top of the directed greybox fuzzer AFLGo [11]. Besides the heuristic used in AFLGo, the *Guidance engine* also takes *separability* (Equation 3.1) into account.

3.5 Evaluation

We evaluate the effectiveness of Fix2Fit in generating test inputs, filtering out overfitted patches and refining patch partitions. Our research questions are as follows:

RQ1 What is the overall effectiveness of Fix2Fit in ruling out overfitted patches?

RQ2 Is Fix2Fit effective for generating crash-free patches?

RQ3 How far can Fix2Fit reduce the pool of patch candidates, if the oracles of only a few (e.g. 5-10) tests are available?

3.5.1 Benchmark Selection

To evaluate our technique, we do not use existing benchmarks for two reasons. First, some existing benchmarks are over-engineered in terms of test-suites. For instance, each subject from the 'ManyBugs' benchmark has 1234 tests, on average. Second, we focus on generating crash-free patches for software crash or vulnerabilities, while most of the defects in existing subjects are logic errors e.g., ManyBugs [63] and Defects4j [53]. Instead, we select a set of real-world subjects from the OSS-Fuzz

Table 3.3: Defect categories of Fix2Fit benchmark

Defect Type	Integer overflow	Buffer overflow	Unknown address
#Defects	29	20	4
Defect Type	Invalid array access	Arithmetic error	Others
#Defects	3	4	21

(Continuous Fuzzing for Open Source Software) dataset[6]. OSS-fuzz, which has recently been announced by Google, is a continuous testing platform for security-critical libraries and other open-source projects. We select projects which contain a large number of bugs and try to reproduce the defects by installing the corresponding versions in our environment. We drop the defects that cannot be reproduced. Furthermore, we focus on subjects which are written in C, since our repair infrastructure works on C programs.

Eventually, we select six well-known open source projects: *Proj.4*, *FFmpeg*, *Libarchive*, *Openjpeg*, *Libssh* and *Libchewing*. Brief descriptions of those projects are given in Table 3.2. Column *#Test* denotes the number of tests from developers accompanying each software project in the OSS-Fuzz repository. For each project, we select a set of reproducible defects based on the above criteria. Column *#Defect* shows the number of selected defects for each project. Totally, 81 unique defects are selected as our subjects. Besides, the bug type of the selected defects is various. Table 3.3 shows the number of defect for each bug type. Specifically, 49 defects are caused by integer overflow or buffer overflow, 7 of them are caused by invalid access, and 25 by arithmetic error or other bugs (e.g. memory leak).

3.5.2 Experimental Setup

To answer *RQ1*, we compare Fix2Fit with AFL[7] and AFLGo[8] based approaches in generating tests to rule out overfitted patches. AFL (AFLGo) based approach constructs candidate patch space using same operators as Fix2Fit, but rules out patches using tests generated by AFL(AFLGo). We choose AFL as our baseline, since it is a fuzz testing which is widely used in industry and academia. AFLGo, a directed greybox fuzzer, can be used for patch testing.

[6]https://bugs.chromium.org/p/oss-fuzz/issues/list
[7]http://lcamtuf.coredump.cx/afl/
[8]https://github.com/aflgo/aflgo

As a post validation process of APR, to determine whether a generated patch is overfitted, people can manually analysis it to check its correctness. To reduce the human burden, our goal is to automatically rule out the overfitted patches as much as possible. Hence, at the patch generation process, it is required to define a overfitting measure. *Opad* [146] proposes a new overfitting measure(*O-measure*), which is built based on the assumption that a correctly patched program should not behave worse than the buggy program. Given a test suite T, suppose:

$\overline{B}$: the set of test cases that make the buggy version pass ($\overline{B} \subset$T)

P: the set of test cases that make the patched version fail(P$\subset$T)

O-measure is defined as the size of $\overline{B} \cap$ P. *Opad* regards a patch as overfit if it has a non-zero *O-measure*, i.e., there are tests from T that pass on the buggy version but fail on the patched version. That is, a patch is regarded as overfitted if it introduces new bugs. In our experiment, we utilize a similar metric, but we change the definition of $\overline{B}$ as the set of test cases that (i) either make the buggy version pass, or (ii) make buggy version crash due to "same" defect as the one we try to fix (by comparing stack trace). The intuition is as follows: if the patched program still crashes due to same defect, we regard the corresponding patch as overfitted patch.

To address *RQ*2, we compare the number of crash-free patches generated by Fix2Fit, AFL and AFLGo-based approach. In our experiment, cross-validation is used to evaluate the crash-free property, where the remaining patches after the filtering of one approach is validated by the tests generated by other techniques. Specifically, suppose (T, P) is a pair of test set and plausible patch set, where the patched program using any patch $p \in P$ does not crash under any test $t \in T$. Let (T_1, P_1), (T_2, P_3) and (T_3, P_3) be the test-patch pairs generated by Fix2Fit, AFL and AFLGo, respectively. We regard $p \in P_i$ as crash-free patch, if and only if the patched program by p does not crash under any test $t \in T_1 \cup T_2 \cup T_3$. Then, we evaluate the percentage of crash-free patches of different techniques.

We answer *RQ*3 by evaluating how many plausible patches can be further ruled out if the newly generated tests are empowered with a few oracles. For any test case which is able to break one partition into several sub-partitions, it finds semantic discrepancies between patches. However, the sub-partitions cannot be ruled out if

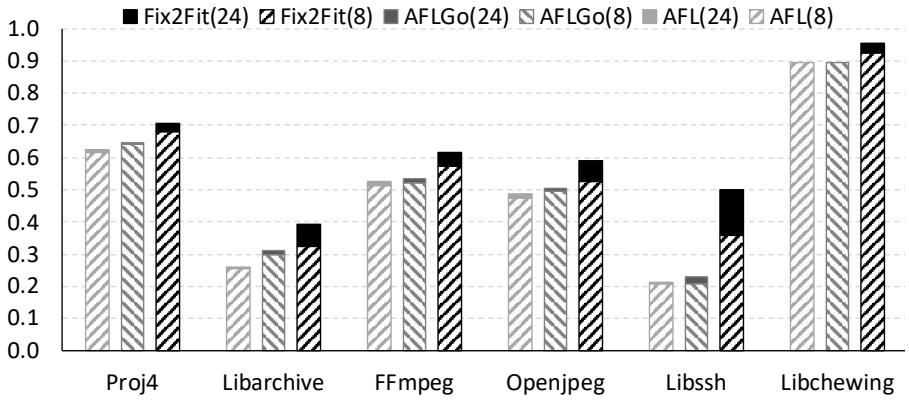


Figure 3.5: Percentage of plausible patches that are ruled out by Fix2Fit

Table 3.4: The averaged $\hat{A}$ of each project with ten runs.

Projects	Proj.4	Libarchive	FFmpeg	Openjpeg	Libssh	Libchewing
$\hat{A}_{12}$	0.70	0.79	0.74	0.68	0.61	0.54

the patched programs do not crash, even though they show different behaviors. We can thus study the reduction in the pool of candidate patches if detailed oracles (such as expected output) for a few (say 5) tests are available. Assuming better oracle of test is given and each subpartition has equal probability to be filtered out, we evaluate the number of patches that can be ruled out (Fig. 3.7).

All the experiments are conducted in the *crash exploration mode*[9] of fuzzer. We start the fuzzing process with the failing test case as seed corpus, and terminate it on timeout. As in state-of-the-art fuzzing experimentation, we set timeout as 24 hours; at the same time we report the effectiveness of our patch pool reduction for smaller values of timeout such as 8 hours. Meanwhile, we set time ($time_x$ in Equation 2.4) to enter "exploitation" as four hours. The experiments are conducted on a device with an Intel Xeon CPU E5-2660 2.00GHz process (56 cores) 64G memory and 16.04 Ubuntu.

3.5.3 Results

RQ1: Effectiveness in ruling out plausible patches. Figure 3.5 shows the percentage of the plausible patches that are ruled out by AFL, AFLGo, and Fix2Fit within 8 and 24 hours, where the percentage of the filtered patches within the first 8

[9]https://lcamtuf.blogspot.com/2014/11/afl-fuzz-crash-exploration-mode.html

hours is marked using diagonal stripes. Note that the AFL-based approach is almost the same as Opad [146], except that we utilize a more precise overfitting measure. For each project, we give the average number of all defects. Compared with *AFL* and *AFLGo*, Fix2Fit rules out more plausible patches for all those six subjects within both 8 and 24 hours. For instance, Fix2Fit filters out 61% plausible patches for FFmpeg, while only 52% of them are ruled out by AFL and 53% by AFLGo within 24 hours. Since fuzzing algorithms involve random decisions, we run each experiment ten times independently and report the Vargha-Delaney statistic measure ($\hat{A}_{12}$) [130] in Table 3.4. Vargha-Delaney statistic is a recommended standard measure for evaluating randomized algorithms [6], which measures the probability that running Fix2Fit rules out more patches than running AFL. Fix2Fit performs better than AFL when $\hat{A}_{12}$ is greater than 0.5. The evaluation results show that Fix2Fit outperforms AFL on all six subjects.

Table 3.5: The averaged number of generated test cases that can rule out plausible patches

Projects	AFL(Opad)	AFLGo	Fix2Fit
Proj.4	4.8	5.9	12.5
Libarchive	11.2	12.8	16.0
FFmpeg	9.8	10.2	13.8
Openjpeg	35.3	35.8	50.3
Libssh	5.1	7.9	8.6
Libchewing	10.7	11.5	11.5

To investigate the reason why Fix2Fit is able to rule out more patches, we give the number of tests generated by each technique that can filter out plausible patches in Table 3.5. On average, Fix2Fit generates 23% more tests that can rule out patches than AFL, and 18% more than AFLGo.

To filter out overfitted patches, fuzzing in Fix2Fit is guided to generate tests that can uncover semantic discrepancies between plausible patches. Therefore, we also evaluate the *patch partition refinement* effectiveness of AFL, AFLGo and Fix2Fit. Figure 3.6 shows the number of generated tests that can refine partitions and the number of patch partitions after refinement. *Origin* is the number of test-equivalence patch partitions with respect to the provided test suite. The histogram represents the number of partitions after refinement, which corresponds to the primary axis (left),

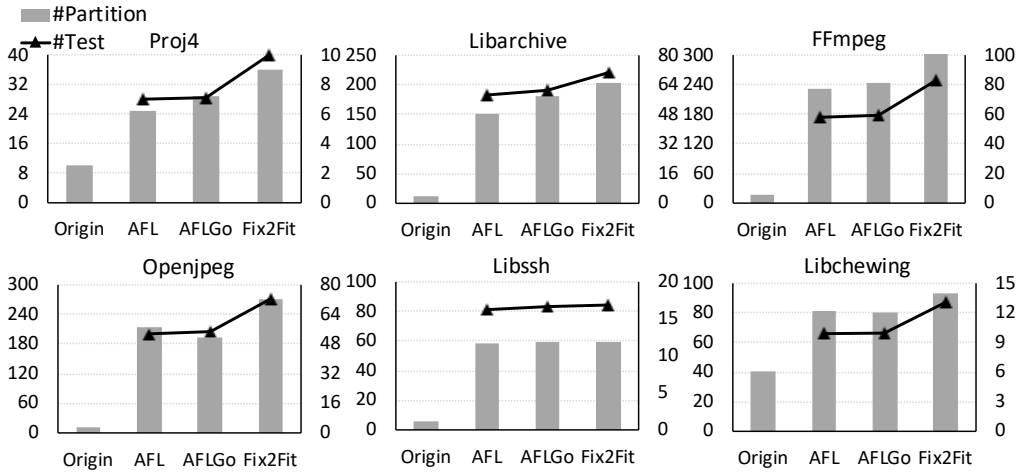


Figure 3.6: Number of patch partitions and generated tests that can break patch partitions.

Table 3.6: Percentage of plausible patches ruled out using *partition refinement (PR)* and *patch filtering (PF)* based heuristic

Projects	Proj.4	Libarchive	FFmpeg	Openjpeg	Libssh	Libchewing
PF	68%	27%	56%	55%	51%	92%
PR	71%	28%	61%	56%	51%	95%

while the line chart shows the number of *partition-refining tests*, which corresponds to the secondary axis (right). Fix2Fit performs better than AFL and AFLGo in both generating *partition-refining tests* and refined partitions. On average, Fix2Fit breaks 34% and 30% more partitions than AFL and AFLGo, respectively.

Although we argue that *partition refinement* is a better heuristic than *patch filtering* for the purpose of guiding fuzzing, we also evaluate heuristics based on *patch filtering*. For *patch filtering* based heuristic, we change the definition of *separability* in Equation 3.1 to

$$separability(t') = \frac{r(t')}{\max_{t \in T_{new}} r(t)} \tag{3.4}$$

where $r(t')$ represents the number of crashing (hence overfitted) patches that are ruled out by test t'. Table 3.6 shows the percentage of patches that are ruled out using the heuristic based on *patch filtering* (*PF*) and *partition refinement* (*PR*). The results show *PR* outperforms *PF* on five subjects and performs equally on one subject.

RQ1: Fix2Fit is able to rule out 18% and 12% more overfitted patches than AFL and AFLGo based approaches.

RQ2: Crash-free patches.

Table 3.7: The percentage of crash-free patches generated by AFL, AFLGo, Fix2Fit

Subject	AFL(Opad)	AFLGo	Fix2Fit
Proj.4	92%	90%	99%
Libarchive	88%	96%	97%
FFmpeg	84%	86%	95%
Openjpeg	82%	85%	91%
Libssh	83%	83%	99%
Libchewing	94%	94%	99%

To fix a bug, new bugs or security vulnerabilities should not be introduced. If one generated test makes the patched program crash, a patch will be directly ruled out. However, since fuzzing does not exhaustively generate all possible tests, the remaining patches may still cause program crashes or introduce new software crashes and vulnerabilities. In this experiment, we evaluate the crash-freedom of patches generated via cross-validation. Based on cross-validation, a crash-free patch should not make the program crash under any test cases generated by any techniques. Table 3.7 shows the percentage of crash-free patches generated by AFL, AFLGo, Fix2Fit. Compared with AFL and AFLGo, our technique significantly improves the percentage of crash-free patches. On average, Fix2Fit generates 96.3% crash-free patches, while 85.4% and 87% patches generated by AFL and AFLGo are crash-free. Especially for Proj.4, over 99.5% patches generated by Fix2Fit is crash-free, compared with 92% of AFL and 90% of AFLGo.

Although most of the patches generated by Fix2Fit are crash-free, there are some patches (3.7%) that cause the program to crash under the tests generated by AFL or AFLGo. Fix2Fit may miss some corner cases since it enters the "exploitation" mode after sufficient "exploration", while AFL and AFLGo keep broadly searching.

RQ2: Fix2Fit could significantly improve the percentage of crash-free patches, and more that 96% patches are crash-free.

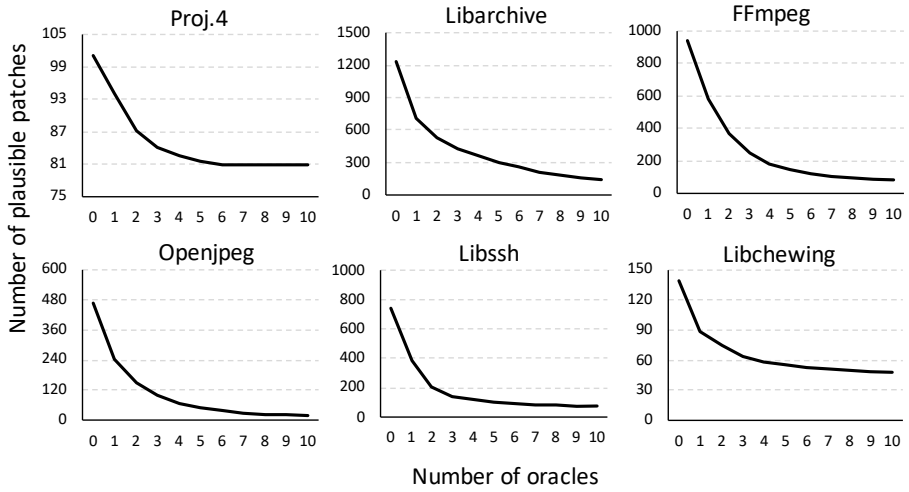


Figure 3.7: Number of plausible patches that can be reduced if the tests are empowered with more oracles

RQ3: Improvement with better oracles. The ability of test cases to filter out overfitted patches is limited by the non-availability of oracles (or expected output) of the generated tests. We also evaluate whether the automatically generated test case can further reduce plausible patches if empowered with better oracles (for at least a few of the generated tests).

Figure 3.7 shows how the number of patch candidates reduces as the number of tests is empowered with oracles. For a test that can break a patch partition into several sub-partitions, we assume only one of sub-partitions is correct if the correct behavior of this test is given. This is because the patch partitions rely on a value-based test equivalence; it is highly possible that only one of the sub-partitions produces an output value same as the expected output. We select the top-10 tests with highest *separability* (heuristic based on *partition refinement*), and collect the number of patches if one, two...ten oracles are given. Generally, the plausible patches for most of the defects can be reduced to a reasonable number. For defects in Openjpeg, the number of plausible patches can be reduced to around 20. In other words, if the oracles of a few tests are available, the pool of candidate patches can be reduced sufficiently so that the remaining patches can be examined manually by the developers.

For the defects which are left with a large number of plausible patches, we are faced with the task of examining these remaining plausible patches. Fortunately,

Table 3.8: Number of remaining partitions after refinement

Projects	Proj.4	Libarchive	FFmpeg	Openjpeg	Libssh	Libchewing
#Partition	4.8	74.4	98.9	47.3	28.3	1.3

developers do not need to examine the remaining patches one by one. They can first find the partition that include the correct patches by examining the patches in the same patch partition together, since they show the same behaviors over all the available tests. Then, only the patches from this partition need to be evaluated one by one to determine the correct patch. Table 3.8 shows the average number of remaining patch partitions after the *partition refinement* by Fix2Fit. The number of remaining partitions, and hence the number of patches to examine, varies between 1-100 in each project. We feel that there might be opportunities for visualization techniques to choose from these remaining $1-100$ patch partitions, using criteria such as syntactic or semantic "distance" from the buggy program.

RQ3: The plausible patches can be reduced to a reasonable number if few tests (<10) are empowered with better oracles.

3.5.4 Threats to Validity

Our current experiments have been conducted for one-line fixes. While an extension of our approach to multi-line fixes is entirely feasible, it can blow up the search space. Second, although we have compared with Opad [146], we could not directly compare with [141, 140] which improve patch quality by test generation; the tools for those approaches are geared to repair Java programs while our repair infra-structure operates on C programs. Finally, our reported results are obtained from the OSS-Fuzz subjects in Table 3.2, and more experiments could be conducted on a larger set of subject programs.

Chapter 4

Alleviate Overfitting via Symbolic Reasoning

Even though test case generation can help discard overfitted patches, it does not provide formal guarantees. To solve this problem, this chapter presents a repair method which fixes program vulnerabilities based on semantic reasoning.

4.1 Introduction

In this chapter, we propose a general approach to combat the overfitting problem in program repair via symbolic reasoning, specifically for fixing *security vulnerabilities*. Our key insight is that information about the underlying cause of vulnerability can be automatically extracted, and the extracted information can then be used to guide Automated Program Repair. The information is extracted in the form of a *crash-free constraint*, representing the abstracted constraint that is violated by the witnessed vulnerability. In order to avoid repeating the vulnerability, the goal of repair is to ensure the *constraint* is always satisfied at the location of the vulnerability.

Challenges Our constraint-driven program repair methodology involves several challenges that need to be overcome.

- *Constraint extraction*: The first challenge is to extract a *crash-free constraint* or CFC from an observable crash/vulnerability. The observable program failure is a concrete property violation when executing a failing test or exploit. In contrast, CFC should capture the properties that all program inputs must satisfy at the crash location, in order to avoid the observed vulnerability.

- *Fix localization*: Our second challenge lies in *fix localization* (FL). *Fix localization* finds one (or more) suitable location(s) to introduce patches. Typically, existing FL approaches, e.g. spectrum-based FL [107], determine fix location according to the execution trace of a large number of test cases. However, a large number of tests are not always available. For an observable vulnerability, there is usually only one test in the form of an exploit. Therefore, it is a challenging task to infer fix locations with only one failing test.

- *Constraint propagation*: After determining the CFC at the buggy location and the fix locations, we then use CFC to guide the patch generation at the fix locations. However, this is not straightforward because the fix locations can be different from the buggy location. For instance, the following code shows a bug where the source and destination of memcpy overlap [1]. The bug is witnessed at line 4, but the correct patch was applied at line 1. We could extract a constraint at the crash location (line 4) to ensure source and destination do not overlap, however, the constraint cannot be directly used to guide patch generation at fix location (line 1).

```
1 - for (i = 3; i < size / 2; i *= 2)  // the correct fix location
  + for (i = 3; i <= size / 2; i *= 2)
2     memcpy (r + i, r, i);
3 if (i < size)
4     memcpy (r + i, r, size - i);    // the crash location
```

  Since fix location(s) could be different from the crash location, the extracted *constraint* must be *propagated* and transformed to guide patch generation at fix location(s). Although a patch at the crash location can also fix this bug (e.g., change the condition at line 3 to `i<size&&size<=2*i`) by disabling the condition to trigger it, such kind of patch actually does not resolve the root cause.

- *Patch synthesis*: The final challenge is to use *program synthesis* to generate candidate patches that ensure that the *constraint* is satisfied for all possible inputs.

Tackling the challenges To address the above challenges, our workflow begins with the detection of an exploitable vulnerability in the form of a *crash*, i.e., unex-

[1] http://www.cplusplus.com/reference/cstring/memcpy

pected program termination due to control flow reaching an invalid state. With the help of sanitizers, such as AddressSanitizer (ASAN) [115] or UndefinedBehaviourSanitizer (UBSAN) [136], we could convert vulnerabilities into normal program crash. In the rest of this chapter, we generally regard a vulnerability as a crash and regard an exploit as a failing test. After witnessing a crash in an exploit, we extract its corresponding CFC (first challenge) using a template-based approach. According to pre-defined templates, a constraint representation of the violated condition—i.e., the *crash-free constraint*—can then be extracted from either the program itself (e.g. user assertion failure), API documentation, or safety properties enforced by dynamic analysis tools such as *sanitizers*. For example, a buffer overflow can be formalized as a violation of constraint:

$$\texttt{access}(\texttt{buffer}) < \mathit{base}(\texttt{buffer}) + \mathit{size}(\texttt{buffer})$$

This constraint is extracted at run-time when the crash is witnessed, and it represents the precise condition that all patched programs must satisfy in order to avoid repeating the same crash. We address the second challenge (*Fix localization*) by examining program dependencies, instead of purely relying on the execution trace of test suites. Specifically, we take as input one failing test, and use the crash location as a starting point and find candidate fix locations using control/data dependency analysis. Once the fix locations are determined, to solve the third challenge (*Constraint propagation*), we *propagate* the extracted constraint backward from the crash location to one or more suitable *fix locations* by calculating the *weakest precondition*. To address the last challenge (*Patch synthesis*), we integrated the second-order program synthesis with counterexample guided inductive synthesis. We synthesize a patch so that the weakest precondition, i.e., the extension of crash-free constraint, cannot be violated, thereby guaranteeing that the patched program cannot repeat the same crash, and thus resolving the vulnerability. Our workflow allows the program repair system to decide between single-line and multi-line fixes as shown by experiments. We instantiate the proposed approach in a prototype named ExtractFix.

Our technique is designed to completely fix security vulnerabilities and alleviate the well-known overfitting problem in program repair [121]. As shown in our evaluation, ExtractFix completely fixes more vulnerabilities, and hence produces

more correct patches, than existing repair techniques.

4.2 Overview

For our purposes, a *crash* is broadly defined to be any program termination due to control flow reaching certain illegal states where conditions/properties are violated. A crash can be caused by the violation of an explicit user assertion (e.g., `assert`(C)), an implicit assertion enforced by the operating system (e.g., illegal memory access), or instrumented check inserted by *sanitizers* to enforce some safety properties. Typical sanitizers, such as AddressSanitizer (ASAN) [115] and UndefinedBehaviorSanitizer (UBSAN) [136], *instrument* the program with implicit assertions that enforce additional properties, such as memory safety, type safety, integer overflows protection, etc. If a sanitizer assertion is violated, the program will abort (i.e., "crash"), usually with an error message indicating the problem. The underlying cause of a "crash" can be automatically extracted in the form of a *crash-free-constraint* (`CFC`). The CFC represents *the condition that should be satisfied at the crashing location in order to avoid repeating the crash.* For example, for a user assertion violation (`assert`(C)), the CFC is C itself, for a `NULL`-pointer de-reference on p the CFC is ($p \neq 0$), and for an array bounds overflow error on $a[i]$ the CFC is ($i < SIZE$) where $SIZE$ is the size of array a. If the crashing program is patched so that the *CFC* is always satisfied at the crash location, then the same crash cannot be repeated for any program input.

Workflow Our basic workflow consists of several components/steps, including:

1. **Constraint Extraction**. Given a program and a single input that exercises the crash, the first step is to extract the "crash-free constraint" (*CFC*). The observable program crash is a concrete property violation when executing a failing test or exploit, while CFC should capture the properties *for all* possible inputs. The CFC is the symbolization or abstraction of the concrete violations. We extract CFC according to predefined templates which formulate the underlying cause of the defect.

2. **Fix Localization**. Once the CFC is generated, one (or more) candidate *fix*

location(s) will be generated using a *dependency-based fix localization* algorithm. Unlike the widely used spectrum-based fault localization (SBFL) [107], we take one failing test as input, and use the crash location as a starting point and find candidate fix locations using control/data dependency analysis.

3. **Constraint Propagation**. The CFC is a constraint over the program state at the crash location. The CFC at the crash location is propagated to a CFC' at a given fix location satisfying the following Hoare triple:

$$\{\{CFC\}'\}\{P\}\{\{CFC\}\} \qquad \text{(CFC-Propagation)}$$

Here, P represents the program between fix location and crash location. CFC' is the least restrictive (weakest) precondition that will guarantee the postcondition CFC [17]. Finding CFC' involves solving CFC-Propagation. For multi-line repair, the approach is generalized and propagation is applied to multiple fix locations.

4. **Patch Synthesis**. Once the fix location and propagated CFC' have been decided, the next step is to generate patch candidates. Patch synthesis involves rewriting the fix location statement ρ into an alternative f such that the following Hoare triple holds:

$$\{true\}\ [\ \rho \mapsto f\ ]\ \{CFC'\}\ P\ \{CFC\} \qquad \text{(CFC-Repair)}$$

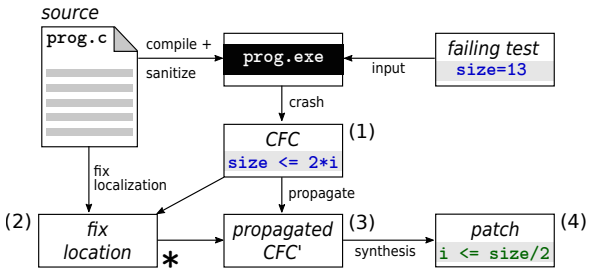
The generated patch $\rho \mapsto f$ fixes the buggy program by substituting expression ρ with the synthesized expression f. The precondition before the patch is set as *true* to ensure the patch fix the bug regardless of the context. The patch is guaranteed to ensure that CFC' is satisfied, meaning that the CFC condition at the crash location cannot be violated in the patched program.

Workflow Example To illustrate our workflow, we consider an example bug from `Coreutils`. The buggy code snippet is shown in Figure 4.1a. Here, the snippet attempts to fill a buffer `r` with a pattern determined by variable *bits* using repeated calls to `memcpy`. The length of each `memcpy` operation is doubled inside the *for*-loop, and the final `memcpy` handles any remaining unfilled space in the buffer. Unfortunately, the code snippet contains a bug [2]. For certain inputs (e.g., `size`=13),

[2]https://debbugs.gnu.org/cgi/bugreport.cgi?bug=26545

```
1  void fillp (char *r, size_t size){
2    ...
3    r[2] = bits & 255;
4    for (i = 3; i < size / 2; i *= 2)
5      memcpy(r + i, r, i);
6    if (i < size)
7      memcpy(r + i, r, size - i);
8  }
```

(a) Buggy code snippet.



(b) ExtractFix workflow overview.

Figure 4.1: Workflow example from Coreutils

the source and destination regions for the final `memcpy` will overlap— an undefined behaviour under the `memcpy` specification. This bug may cause a program crash on some platforms. Specifically, when `size`=13, the *for*-loop will terminate in the second iteration with i=6 and $size/2$=6 (integer division). Then, at line 7, the source and destination of memcpy overlap because $r+(13-6)>r+6$. Using an appropriate sanitizer (UBSAN), this program will crash on the final `memcpy` call.

Figure 4.1b shows the overall workflow of our approach. We start with the single crashing input (`size`=13) that triggers the crash on line 7 (highlighted). Step (1) generates the *CFC* corresponding to the crash according to a predefined template. The *CFC* template (shown in Section 4.3.1) of `memcpy(p, q, s)` is defined as $\texttt{p}+\texttt{s} \leq \texttt{q} \vee \texttt{q}+\texttt{s} \leq \texttt{p}$. In this case, CFC is

$$(\texttt{r}+\texttt{i}+\texttt{size}-\texttt{i} \leq \texttt{r} \vee \texttt{r}+\texttt{size}-\texttt{i} \leq \texttt{r}+\texttt{i}) \equiv (\texttt{size} \leq 0 \vee \texttt{size} \leq 2*\texttt{i})$$

Since size is an unsigned integer (size_t) value, we only focus on the second clause $\texttt{size} \leq 2*\texttt{i}$ in this example. Step (2) determines candidate fix locations. One promising fix location is the *for*-condition on line 4 (highlighted) since there exists a control dependency with an assignment (`i *= 2`, line 4) that has a data dependency with the crash location. Step (3) propagates the *CFC* to the fix location along all feasible paths. In this case, the *CFC* is propagated along one path with path constraint $i<size$, and *CFC* remains unmodified. Step (4) synthesizes a patch f to replace the *for*-condition. To completely fix the bug, we should ensure $\texttt{size} \leq 2*\texttt{i}$ is always satisfied after applying f. In this case, the synthesizer gives $i <= size/2$. Thus, the program can be patched as follows:

```
- for (i = 3; i < size / 2; i *= 2)
+ for (i = 3; i <= size / 2; i *= 2)
```

The resulting patch is equivalent to the developer patch. In contrast, test-driven program repair approaches may produce overfitting patches. For example, the following patch generated by Fix2Fit [34] fixes the bug for `size`=13, but does not generalize to other crashing inputs, e.g. `size`=7.

```
+ for (i = 3; i < size / 2 || i == 6; i *= 2)
```

4.3 Methodology

Our workflow for program repair involves constraint extraction, propagation, and patch synthesis. In this section, we discuss each step in more details.

4.3.1 Crash-Free Constraint Extraction

Our workflow begins with a vulnerable program and a single crashing input. The first step is to extract both (1) the *crashing location* (e.g., filename/lineno), and (2) the *crash-free constraint* (*CFC*) representing the condition that was violated and the underlying cause of the crash. For (1), the crash location is extracted according to debugging information when the crash is triggered, meaning that the program must be compiled with debugging enabled (`-g`). For (2), the *CFC* extraction is *template*-based, and is instantiated from the crashing expression/statement. Our repair technique currently considers crashes due to:

(1) *Developer*-induced crashes, i.e., `assert`(C) failure;

(2) *Sanitizer*-induced crashes caused by the program violating a sanitizer-enforced *safety property* (e.g., memory safety, type safety, etc.);

A summary of the different kinds of crashes and the corresponding *CFC*-templates are shown in Table 4.1. Here, the *crash expression* is matched against the corresponding crashing expression/statement from the buggy program, and the *CFC*-template is instantiated accordingly. We choose those templates because they cover the common errors and vulnerabilities in C/C++ programs, e.g. null pointer dereference,

Table 4.1: Basic crash classes, crash expressions/statements, and corresponding *Crash-Free Constraint CFC*-template. We consider seven types of crash: explicit developer assertion violation, sanitizer-induced crash such as buffer overflows/underflows, integer overflows, API constraint violation.

Class	Template ID	Expression	*CFC* Template
developer	T_1	`assert`(C)	C
sanitizer	T_2	`*p`	`p+sizeof(*p)` $\leq$ *base*(`p`)+*size*(`p`) `p` $\geq$ *base*(`p`)
	T_3	`a` *op* `b`	`MIN` $\leq$ `a` *op* `b` $\leq$ `MAX` (over $\mathbb{Z}$)
	T_4	`memcpy(p, q, s)`	`p+s` $\leq$ `q` $\vee$ `q+s` $\leq$ `p`
	T_5	`*p` (for `p`=0)	`p` $\neq 0$
	T_6	`a / b` (for `b`=0)	`b` $\neq 0$

integer/buffer overflow. In this thesis, we restrict to fix the bugs supported by these templates. Our tool can also fix other kinds of bugs by extending the templates.

For Example 4.2, the crashing statement `memcpy(r+i, r, size−i)` is matched against the template from Table 4.1 using the substitution `p=r+i`, `q=r`, and `s=size−i`. This yields the following *CFC* after substitution and simplification:

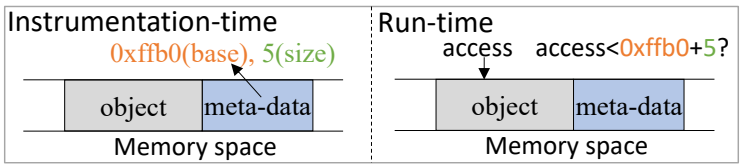
$$(\texttt{r+i+size−i} \leq \texttt{r} \vee \texttt{r+size−i} \leq \texttt{r+i}) \equiv (\texttt{size} \leq 0 \vee \texttt{size} \leq 2*\texttt{i})$$

We now discuss the *CFC* generation step in more details.

User-Assertion. The *CFC* for user assertions is relatively straightforward to generate. Assuming the crash is caused by a *user assertion failure* `assert`(C), the *CFC* can be read directly from the assertion statement itself, i.e., *CFC*=C.

Sanitizer Constraint Extraction For our purposes, a *sanitizer* is any dynamic analysis tool that instruments/modifies the program with additional runtime checks enforcing certain *safety properties*, such as memory safety, preventing integer overflows or other undefined behavior avoidance. Typically, sanitizers insert instrumented checks/assertions before relevant operations. For example, as shown in the following figure, the instrumentation (left part) of most spatial memory safety sanitizers (a.k.a., bounds-check sanitizers) track *object bounds information* (i.e., the *size* and *base* address of each allocated object) using a disjoint metadata store or related method. At run-time (right part of the following figure), this metadata is used to look up the object bounds corresponding to the dereferenced pointer, and this

pointer (*access*) is checked against these bounds (*base+size*). If the instrumented check fails, the program is terminated, i.e., "crashes". Crashes can be caused by



hardware failure such as `NULL`-pointer dereference and divide-by-zero are detected using an appropriate santizer or *signal handler*, e.g., `SIGSEGV` with `si_addr`=0 and `SIGFPE` with `si_code`=`FPE_INTDIV` respectively. The corresponding *CFC* ensures that the crashing symbolic pointer/divisor is not zero.

Sanitizers can only detect "crashes" on concrete program state, e.g. specific values of *size* and *base* on a certain test. We then symbolize the safety condition that sanitizer enforces by mapping the concrete state back to variables/memory relevant to the crash. For the example in Figure 4.1a, a sanitizer detects source/destination memory regions overlap when *size=13*. We then generate *CFC* by mapping the concrete value of source/destination back to program variables r and $r+i$, respectively. To map a concrete crashing state back to symbolized variables, we extend the metadata by also restoring the corresponding program variable information (e.g. variable name, type) representing *size* and *base*. When the crash is detected, we can simply construct the crash-free constraints using the symbolized program states (program variables). However, in some cases, we may fail to symbolize constraints because some variables used to construct *CFC* are not accessible at the crashing points, i.e. the variables stored in metadata have already been killed at the crashing points. For example, a buffer is dynamically allocated with a local variable in one function as buffer length, while the error is observed in a different function. Therefore, when generating *CFC*, the variable representing buffer length is not available at the crash location. In this case, we could symbolize the *CFC* using an *extended program state*.

Sanitizer Constraint Language In the context of program verification, ghost codes are a set of auxiliary codes that are added to the original program for the purpose of verification [28]. Ghost code does not interfere with regular code, in the sense that it can be erased without observable difference in the program outcome.

Inspired by ghost codes, we insert a set of auxiliary codes to help us produce patches at the fix location. Specifically, we regard the sanitizer-inserted *extended state* as auxiliary codes. The *extended state*, that is managed by a runtime library, is used by Sanitizers to check program properties. For instance, AddressSanitizer uses function the `sizeof(buffer)` from the runtime library to detect buffer overflow. This extended state is not part of the original program itself, and it is only used to help to check program properties. We also allow the generated *CFC* to include such extended states, including functions/types/variables that do not necessarily appear in the original program. For example, in the case of bounds-check sanitizers, we introduce two new abstract functions:

- *base*(`p`): the base address of the object referenced by `p`; and
- *size*(`p`): the size (in bytes) of the object referenced by `p`.

The generated *CFC* will be over these extended functions (see Table 4.1). Another example is integer-overflow sanitizers, where the generated *CFC* (e.g., `a+b` $\leq$ `MAX`) is over arbitrary precision integers ($\mathbb{Z}$) rather than the original 32bit integer type. With the help of the extended language based on instrumentation, our technique is able to produce *CFC* regardless of the scope of the referenced variables. For the purpose of *CFC*-generation, we utilize a sanitizer-like instrumentation and extract the extended-language constraints "as-is", and defer further simplification/handling to the latter stages of our workflow.

4.3.2 Dependency-Based Fix Localization

Once the crash location and *CFC* have been determined, the next step is to decide one (or more) *fix location(s)* where the patch(es) are to be applied. Typically, existing FL approaches, e.g. spectrum-based FL [107], find candidate fix locations by analyzing the execution trace of passing and failing tests. The FL results depend on the quality of the tests, but high-quality tests are not always available. Unlike traditional FL approaches, we make a minimal assumption that only one failing test (exploit) is available, which is a very common scenario when security vulnerabilities are found.

Algorithm 3: Fix localization algorithm

Input: A crash location (*crashLoc*) and an *Inter-procedure Control Flow Graph* (*ICFG*)
Output: A set of candidate fix locations (*fixLocs*)

```
1  fixLocs := {crashLoc};
2  repeat
3  |   fixLocsPrev := fixLocs;
4  |   foreach fixLoc ∈ fixLocsPrev, loc ∈ ICFG − FixLocsPrev do
5  |   |   if depends(loc, fixLoc) ∧ dominates(CFG, loc, crashLoc) then
6  |   |   |   fixLocs := fixLocs ∪ {loc};
7  |   |   end
8  |   end
9  until fixLocsPrev = fixLocs;
10 rFixLocs := rank(fixLocs);
11 return rFixLocs;
```

The main intuition of our dependency-based fix localization is that the fix location(s) ought to exhibit a *control* or *data*-dependency with the crash location, such that, the statement at fix location can influence the truth value of the *CFC*. We are also looking for fix location(s) which appear on the execution path of the crashing test. As a practical realization of these intuitions, our repair technnique uses the *crash location* as the starting point and performs backward *control* and *data*-dependency analysis along with crashing path. Algorithm 3 summarizes the *fix localization* algorithm to decide candidate fix locations. Here, the algorithm takes as input an *Inter-procedure Control Flow Graph* (ICFG) and a *crash location* (*crashLoc*). Since the ICFG may be large in practice, partial ICFG is constructed by considering locations visited by the failing test (exploit) and dependency analysis is performed with the crashing statement as the slicing criterion. The algorithm iteratively builds a set of potential *fix locations* (*fixLocs*) by adding nodes that (1) have a (transitive) dependency with the crash location, and (2) *dominate* the crash location. Finally, the algorithm generates a sequence of *fix location* candidates, which are ranked according to the distance to the crash location.

Dependency Closure Our algorithm also considers the *transitive closure* of static data and control dependencies [127] of the crashing statement to compute potential fix locations. Data dependencies are determined using the standard *def-use-chain* traversal algorithm over a *Single Static Assignment* (SSA) representation of the

program. We detect control dependencies using the standard *Control Dependence Graph* (CDG) [18] program analysis as part of the LLVM compiler infrastructure. Considering Figure 4.1a once more, the *for*-condition (line 4) is a control dependency on the assignment statement (`i *= 2`, also line 4), and the crash location (line 7) is data dependent on this assignment. Thus, the *for*-condition is a potential fix location.

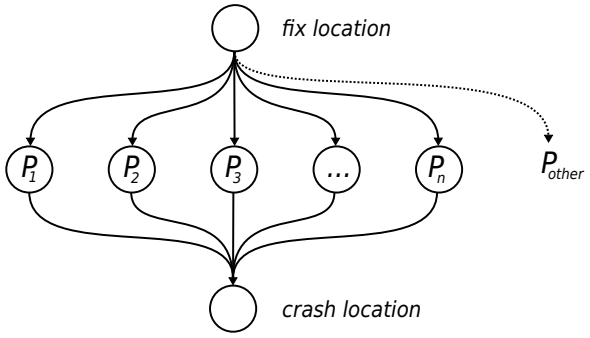


Figure 4.2: Illustration of the fix localization algorithm. The algorithm attempts to find a node (*fix location*) that (1) is a dependency of, and (2) dominates the (*crash location*). All paths from the entry point to the crash location must pass through the fix location. There can be more than one path (P_i) between the fix and crash locations. It is allowable that some paths, including loops, from the fix location do not pass through the crash location (P_{other}).

Crashing Path and Dominance The set of all (transitive) data and control dependencies of the crash location can be quite large, leading to many potential fix locations. To reduce the number of potential fix locations, we restrict the fix location(s) should exist somewhere along the concrete path belonging to the original crashing test case. Furthermore, in order to guarantee that the patched program satisfies the *CFC*, our fix localization algorithm only considers statements that *dominate* the crash location—i.e, all paths from the entry point to the crash location must also pass through the fix location, as illustrated in Figure 4.2. Considering Example 4.2, the *for*-condition (line 4) *dominates* the crash location, since all paths from the entry will visit the *for*-condition at least once. There are usually multiple nodes that dominate the crash location in real-world programs, meaning there are multiple potential fix locations. Note that, there are always at least two nodes that dominate the crash location: the entry point, and the crash location itself.

4.3.3 Crash-Free Constraint Propagation

The weakest precondition of a formula φ is the least restrictive precondition that will guarantee φ [17]. We consider the problem of backward propagation as finding the weakest precondition CFC' at fix location l that necessarily drives the program to the crash location and satisfies CFC at the crash location. As shown in [45] (Theorem 9), for all deterministic programs P and any desired post-condition Q: $wp(P,Q)$=$fwd(P,Q)$, where wp represents the weakest precondition that drives program P to satisfy Q, while fwd is the result generated by forward symbolically executing P from the first statement to the last and substituting the used variables in Q with symbolic state of variables.

Example 4.3.1 *Consider the following program P:(x=x+x; x=x+x; x=x+x) and post-condition Q: x<8, the weakest precondition to guarantee Q is $wp(P,Q) = \{x < 1\}$. Similarly, if we set x to be a symbolic variable and symbolically execute P from the beginning, we would get $8x < 8$. That is, $fwd(P,Q) = \{x < 1\}$.*

In this chapter, we use forward symbolic execution to calculate the weakest pre-condition. Given a fix location l, crash location c, and CFC, we perform symbolic execution between l and c, and calculate the weakest precondition CFC' at l. Our symbolic execution starts concrete execution with a concrete input t until the fix location l. The concrete input t can be the exploit of the vulnerability, or any test that can drive the program to l. From the fix location, we insert symbolic variables and start symbolic execution to explore all the paths Π from fix location l to crash location c.

Symbolic Variable Insertion At fix location, existing semantics-based repair techniques, e.g. Semfix [93], Angelix [79] and [76], represent the to-be-repaired expression as (either a first-order or a second-order) symbolic variable. Symbolic execution captures the constraint of passing a given test suite T by exploring alternate paths from the fix location along which the execution of T could be driven in the fixed program. In contrast, in our approach, symbolic execution computes the weakest pre-condition of the crash-free constraint CFC, by exploring *all* paths between fix location and crash location. We apply the following transformation schemes to introduce second-order symbolic variable ρ:

- changing the right-hand side of an assignment:

$$x := E; \mapsto x := \rho(v_1, \ldots, v_n);$$

- changing a condition:

$$if(E)\{\ldots\} \mapsto if(\rho(v_1, \ldots, v_n))\{\ldots\}$$

- adding an if-guard:

$$S; \mapsto if(\rho(v_1, \ldots, v_n))\{S;\}$$

- adding an if-return:

$$insert : if(\rho(v_1, \ldots, v_n))\{return\ C;\}$$

where S is statement, E is expression, C is constant and $v_1, \ldots, v_n$ are the live variables at the fix location. We use if-return transformation only if the others fail to generate a correct patch, and the error handling code C is generated using Talos [43]. Apart from generating a (second-order) symbolic variable ρ at the fix location (to capture the to-be-synthesized expression) we also set the live variables V (on which CFC is dependent) as symbolic variables. We might introduce multiple symbolic variables. If a variable v at the fix location may affect the truth value of CFC at the crash location, we will set v as a symbolic variable. This strategy introduces a minimal number of symbolic variables while ensuring that all relevant paths between fix and crash location are explored. With these symbolic variables, we can explore and navigate the paths between fix and crash location.

Symbolic execution scope To avoid exploring irrelevant paths, all the paths that never reach crash location, e.g. P_{other} in Figure 4.2, are terminated early (whether a path can reach c is determined by analyzing control flow graph). With the help of symbolic variable injection and early termination, the explosion of paths is reduced. Furthermore, since fix locations are usually close to the crash location, we can further alleviate the path explosion problem which is common in symbolic execution.

Constraint collection After symbolic exploration, we collect the path constraints pc_j for each path $\pi_j \in \Pi$ (all feasible path from l to c). Besides, following each π_j, all the variables used in *CFC* can be represented using the symbolic variables (V and ρ). By replacing the elements in *CFC* with the symbolic representations of V and ρ, we rewrite *CFC* as CFC'_j. Then, $pc_j \Rightarrow CFC'_j$ will be exactly the same as the constraint by backward propagating *CFC* from crash location to fix location along path π_j. Consider the following program

$$input\ x,\ i;\quad if(i{>}0)\ y{=}x{+}1;\ else\ y{=}x{-}1;\quad output\ y;$$

Suppose the *CFC* is $(y > 5)$, along the *if-then* branch, we will get the constraint $(i{>}0 \Rightarrow x + 1 > 5)$.

Constraint Simplification (Optional) The propagated constraints may still contain extended sanitizer-supplied functions (e.g., $base(p)/size(p)$) or types (e.g., $\mathbb{Z}$ for integer overflow). There are two basic approaches to handling the extended constraint language: (1) Synthesize the patch "as-is". If necessary, extra functionality can be supplied using a suitable runtime library; or (2) translate the extended constraints into the native language if possible.

Approach (1) is the most general. For example, runtime implementations of the $base(p)/size(p)$ are available using a suitable *library*, meaning these functions can be used in a patch. The downside is that this introduces an additional dependency on the patched program, which may be undesirable for some applications. The alternative (2) approach is to rewrite the extended constraints back into the native language if possible. For example, using a simple static analysis, our tool searches for a dominating CFG node where the object associated to p is first allocated, e.g., ptr=`malloc`(len). If such a node is found, then our tool can substitute $base(p){=}ptr$ and $size(p){=}len$. This approach is less general than (1) since it depends on a suitable substitution to be found.

Note that the weakest precondition calculation inherits the limitation(s) of how symbolic execution is performed. For instance, we may generate incomplete weakest preconditions if there are loops between fixing location and crash location. In our setting, we also inherit the solution from symbolic execution by adding a bound to the number of loop iterations, which may result in incorrect patches as shown in

Section 5.

4.3.4 Patch Synthesis

After backward propagation of crash-free constraints, patch synthesis is used to rewrite the statement at fix location and guarantee:

$$\{true\}[\ \rho \mapsto f\ ]\{CFC'\}$$

Although our reasoning is performed on a partial program (from fix to crash location), the synthesized patch will be also effective for the whole program, because the precondition (*true*) is applied. Once $\{true\}[\rho \mapsto f]\{CFC'\}$ is satisfied, CFC' is guaranteed to hold under any context.

Instead of satisfying input-output relations as shown in Equation 2.3, the synthesizer is used to produce a patch satisfying a certain constraint. Suppose Π is the set of feasible paths between fix and crash location, for each path $\pi_j \in \Pi$, the generated patch f should imply CFC'_j under **all** input space. Then, we change the definition of $\varphi_{correct}$ defined in Section 2.2 to:

$$\varphi_{correct} \coloneqq \bigwedge_{j=1}^{|\Pi|} \left(\left(\rho = f(V) \wedge pc_j \right) \Rightarrow CFC'_j \right) \tag{4.1}$$

where f represents the to-be-synthesized function and V is the set of variables used by f. For the example 4.2, $\varphi_{correct}$ will be:

$$\varphi_{correct} = (\rho = f(size, i) \wedge \neg\rho \wedge i < size) \Rightarrow size \leq i * 2$$

Since f is a function and the implication should hold **for all** inputs, φ_{correct} is actually a second-order formula. To solve this formula, ExtractFix uses the idea of *second-order solver* [76] to convert φ_{correct} to a first-order formula, and then uses *counter-example guided inductive synthesis (CEGIS) [48]* to find proper patches. By synthesizing f satisfying $\varphi_{correct}$, we can handle all bug-triggering inputs that violate CFC', hence CFC.

Though the generated patch makes CFC hold, we may still have a wide choice of candidate patches. For fixing the bug in example 4.2, several patches satisfying the $\varphi_{correct}$ (equation 4.1) could be generated, such as $\{1, i \leq size/2\}$. Obviously, the second one is more likely to be correct. To further improve the quality of patches,

Algorithm 4: Extension of second-order synthesizer

Input: The original buggy expression e, the constraint $\varphi_{correct}$
Output: A patch f which satisfies $\varphi_{correct}$

```
1  hard := φ_wfp;
2  soft := φ_syn ;
3  patches := ∅ ;
4  while |patches| ≤ N and (timeout not reached) do
5  |   f_c := pMaxSMT(hard, soft) ;
6  |   I := SMT(¬φ_correct[f ↦ f_c]) ;
7  |   if I ≠ None then
8  |   |   hard := hard ∧ φ_correct[V ↦ I] ;
9  |   else
10 |   |   patches := patches ∪ {f_c} ;
11 |   end
12 end
13 return semSelect(patches) ;
```

the intuition is that the correct patch should be similar, both syntactically and semantically, with the original program. To generate "similar" patches, ExtractFix extends the second-order solver proposed by Mechtaev, Griggio, Cimatti, and Roychoudhury by further considering the distance between the patched and original program.

The overall workflow of our synthesizer is shown in Algorithm 4, which takes as input the suspicious expression e and $\varphi_{correct}$, and generates a patch f. ExtractFix first generates a patch candidate by solving combined hard and soft constraints using MaxSMT[31] (line 5 of Algorithm 4). The hard constraint is initialized as φ_{wfp} (refer section 2.2), which ensures the candidate is well-formed. The soft constraint φ_{syn} formulates the syntax distance between buggy expression e and candidate patch. More formally, we build abstract tree T_e for e, and T_c for the patch candidate, and define

$$\varphi_{syn} := \bigcup_{k=1}^{|T_e|} \left\{ T_e^k == T_c^k \right\} \tag{4.2}$$

where T_e^k (T_c^k) denotes the k-th node of tree $T_e(T_c)$. MaxSMT constructs a patch candidate f_c which strictly satisfies the hard constraint, and satisfies the maximum number of soft constraints (shortest distance). The candidate f is then validated by Satisfiability Modulo Theories (SMT) solver [90] to check whether an input that violates $\varphi_{correct}$ exists (line 6). If such a counter-example I exists (line 7-8), the counter-example I is first encoded into first-order logic and then added into the

hard constraint. Consider the example shown in Figure 4.1a, in the first iteration, assume $f_c = \lambda i.\ \lambda size.\ i{<}size/2$, then

$$\varphi_{correct} = (\rho = (\lambda i.\ \lambda size.\ i < size/2) \wedge \\ \neg\rho(i, size) \wedge i < size) \Rightarrow size \leq i * 2$$

is violated when $i = 6$ and $size = 13$. Therefore, we add

$$(\rho = f \wedge \neg\rho(6, 13) \wedge 6 < 13) \Rightarrow 13 \leq 12$$

i.e. $f(6, 13) = true$, into the hard constraints. With the refined hard constraints, the candidate f_c generated in the next iteration will ensure $\varphi_{correct}$ must be satisfied under I, i.e. $f_c(6, 13) = true$. Eventually, a plausible patch f_c is thereby generated, which will be added into the *patches* list (line 10). The process continues until *timeout* is reached or we find N plausible patches, where *timeout* and N are defined by users.

The patch synthesis of ExtractFix is built on top of SE-ESOC [76]. The difference with SE-ESOC is two-fold. First, we introduce a set of soft constraints (Equation 4.2) to formulate the distance between original expression and patch candidates. Such that, we can generate patches that are syntactically similar to the original program. Second, SE-ESOC is designed to solve a problem with existential quantifiers, i.e., generate patches to pass **existing** tests. In contrast, the synthesis in ExtractFix generates patches that fix the bug **for all** the valid inputs. Thus, we integrate counterexample guided inductive synthesis into SE-ESOC to make it support solving problems with universal quantifiers, i.e., generate patches to fix the bug under all input space.

Among N plausible patches, the most likely to be the correct one is selected according to its semantic distance to the origin buggy expression e (*semSelect* line 13). Specifically, we (1) generate a set of inputs In that can distinguish plausible patches in terms of their semantics (2) for each $in \in In$, calculate the values of each plausible patch and expression e (3) calculate the value distance between each patch with e (4) select the patch with the shortest distance.

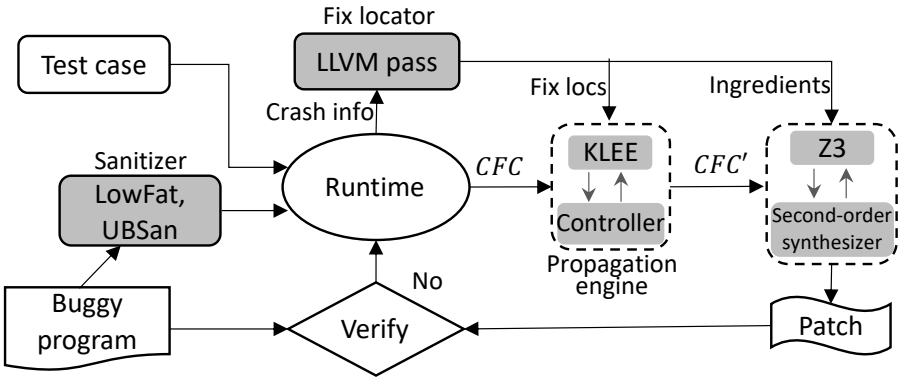


Figure 4.3: The architecture of ExtractFix.

4.3.5 Multiple-Line Fix

The proposed work-flow can be easily extended to support bug-fixing in multiple locations. Fix localization can be generalized as a set of nodes that collectively dominate the crash location, i.e., all paths must go through one of the nodes from the set. Suppose we are introducing patches at location $\{l_1, \ldots, l_n\}$, when propagating CFC, multiple second-order variables $\{\rho_1, \ldots, \rho_n\}$ are introduced to represent the to-be-synthesized expressions at $\{l_1, \ldots, l_n\}$, respectively. Correspondingly, the generated CFC' will involve multiple second-order variables $\{\rho_1, \ldots, \rho_n\}$. Then, the goal of synthesizer is to generate a set of function $\{f_1, \ldots, f_n\}$ to satisfy:

$$\varphi_{correct} \coloneqq \bigwedge_{j=1}^{|\Pi|} \left(\Big(\bigwedge_{i=1}^{n} \big(\rho_i = f_i(V_i)\big) \wedge pc_j \Big) \Rightarrow CFC'_j \right) \tag{4.3}$$

4.4 Implementation

We have implemented our approach in a tool named ExtractFix, whose architecture is shown in Figure 4.3. ExtractFix takes as input the vulnerable program, exploit (test case) and produces patches. ExtractFix is composed of four main components: *constraint extractor*, *fix locator*, *propagation engine* and *patch synthesizer*.

Constraint extractor takes as inputs the vulnerable program and exploit, generates a crashing location, and a crash-free constraint CFC. The constraint extractor is mainly implemented on top of sanitizers: Lowfat [21, 22] for buffer overflow/underflow and UBSAN [136] for integer overflow, null pointer dereference

and etc. Although our prototype supports a specific set of defects, other bugs can be supported by integrating new sanitizers and corresponding templates. Once a crash is detected, the concrete crash condition is symbolized into crash-free constraint CFC by mapping the concrete value back to program variables. To enable the mapping, the programs should be compiled using clang with the debug option.

Fix locator takes as inputs the buggy program and crash information, and produces a set of ranked fix location candidates. The *fix locator* is a static analysis tool and is implemented as an LLVM pass [3]. We implement it on top of LLVM because LLVM provides a set of interfaces to generate control flow graphs and data dependency graphs.

Propagation engine is built on top of KLEE [13]. For the purpose of generating the weakest precondition, we modify the path exploration of KLEE in the following two aspects. First, we change the constraint collection by only considering the path constraints between fix and crash location. Second, we early terminate the paths that cannot reach a crash location. The execution scope is controlled by *Controller*.

Patch synthesizer is a second-order synthesizer which is implemented according to the approach proposed in [76]. Besides, ExtractFix implements three new features: (1) taking the CFC as correctness criterion (2) combining with counter-example guided synthesis and (3) taking into account the distance between patches and original buggy expression. In our implementation of the synthesizer, we use Z3 [90] as a backend SMT solver.

4.5 Evaluation

We evaluate the effectiveness and efficiency of ExtractFix and answer the following research questions.

RQ1 What is the overall effectiveness of ExtractFix in fixing vulnerabilities?

RQ2 Compared with state-of-the-art techniques, can ExtractFix alleviate the overfitting problem in automated program repair?

RQ3 What is the efficiency of ExtractFix in generating patches?

[3]LLVM Pass: http://llvm.org/docs/WritingAnLLVMPass.html

4.5.1 Experimental Setup

We evaluate our approach on two sets of benchmarks: ManyBugs [63] and our own constructed benchmark. ManyBugs is a C program benchmark suite that is widely used to evaluate automated program repair techniques, such as GenProg [64], Prophet [70] and Angelix [79]. Since we are focusing on vulnerabilities, we only select bugs that relate to vulnerabilities as our subjects. We therefore select our subjects based on the following criteria:

(1) we only consider bugs related to vulnerabilities, including segmentation fault, buffer overflow/underflow, integer overflow;

(2) the target application can be compiled into LLVM [57] bitcode and executed by KLEE [13];

(3) the target vulnerability can be reproduced in our environment.

We omit two applications of the benchmark (python and fbc) because they used some intrinsic functions (e.g., `fabs`) that KLEE does not support. In total, we select 26 defects from three applications: Libtiff, Lighttd, and Php.

Table 4.2: The subject programs and their statistics

Program	#Vul	Loc	Description
Libtiff	11	81K	library for processing TIFF files
Binutils	2	98K	a set of programming tools for creating and managing binary programs
Libxml2	5	299K	XML C parser and toolkit
Libjpeg	4	58K	C library for manipulating JPEG files
FFmpeg	2	617K	library for processing audio & video
Jasper	2	29K	library for coding & manipulating image
Coreutil	4	78K	GNU core utilities
Total	30	—	—

Besides ManyBugs, we also constructed an additional vulnerability benchmark suite from a set of popular applications by searching the online databases [134, 133, 135]. Those databases provide a list of entries, and each of them contains an identification number, a short description of the bug and optional reproducer (i.e. exploit). We obtain our candidate bugs by searching for the bug types

(including buffer-overflow/underflow, integer-overflow, divide-by-zero, null pointer, and developer assertion) that our prototype supports. We just consider the bugs reported after 2010 because the earlier bugs are harder to reproduce. Then, we randomly select and manually filter the subjects based on the following four criteria:

(1) exploit(s) to trigger the vulnerability is available or exploit(s) can be constructed from the available information;

(2) the target vulnerability has already been fixed by developers so that we have the ground truth on how to fix it;

(3) the target application can be compiled into LLVM [57] bitcode and executed by KLEE [13];

(4) the target vulnerability can be reproduced in our environment.

Finally, 30 unique vulnerabilities across seven applications are selected as our benchmark, which includes 16 buffer-overflow/underflow, 4 integer-overflow, 5 divide-by-zero, 3 API assertion, and 2 null pointer dereference. The exploits, as well as the instructions to reproduce the bugs, are obtained from blogs of researchers, bug reports, exploit databases or the attachments along with patch commit. The selected subjects are across seven applications, and their brief descriptions are given in table 4.2. Column *Loc* represents their lines of source code, while column *#Vul* shows the number of selected vulnerabilities for each application. The main difference between ManyBugs and our own constructed benchmark is that the subjects from ManyBugs include a huge number of test cases, while the subjects in our benchmark only have an exploit and few developer test cases. Note that ExtractFix is designed for working with a few cases.

The experiment is directly conducted on these vulnerable applications on a device with Intel Xeon CPU E5-2660 2.00GHz process (56 cores) 64G memory and 16.04 Ubuntu. We set timeout for the symbolic execution and program synthesis as 30 minutes each. Note that, we do not support parallelism yet. All the results are generated using sequential algorithms.

4.5.2 Experimental Results

4.5.2.1 Effectiveness of ExtractFix in fixing vulnerabilities

To answer RQ1, we evaluate the effectiveness of ExtractFix in the following three aspects: 1) extracting CFC 2) finding fix locations and 3) generating patches to fix vulnerabilities. Recall that the vulnerabilities are formalized as violations of constraints, we first evaluate whether ExtractFix can successfully extract such constraints for the given vulnerabilities. For the generated constraint, we generate the ground truth of correctness by manually analyzing the source code and root cause of the vulnerability. For instance, we manually analyze the condition that a buffer overflow can be triggered, and check the correctness of CFC. Given CFC, we then evaluate whether ExtractFix can find the correct fix locations by referring to the developer patches. As our dependency-based fix localization creates a set of ranked candidate fix locations, we retrieve how many candidates we need to inspect until we hit the correct one. A fix location l is correct if we can generate semantically equivalent patches at l with developer patches. Given CFC and fix location candidates, we then evaluate the effectiveness of ExtractFix in generating fixes, and compare with existing automated program repair tools: Prophet [70], Angelix [79] and Fix2Fit [34]. Prophet is a search-based automated program repair tool, which ranks patch candidates using a machine learning-based approach. In our experiment, we are using the pre-trained model released by the authors of Prophet. Angelix is a state-of-the-art semantic program repair tool, which extracts patch constraints from test cases and then directly synthesizes a patch. Fix2Fit proposes to generate additional test cases to filter out the overfitted patches. Since Prophet and Fix2Fit are all test-driven program repair tools, we run all those tools with test cases which are composed of 1) exploit that can trigger the vulnerability and 2) available developer tests. Note that, except for one exploit, ExtractFix does not need the additional test. As optional post-processing, developer tests could be used to verify the correctness of patches generated by ExtractFix. All the generated CFC, fix locations and patches can be found in `https://extractfix.github.io`

Table 4.3 shows our evaluation results. The first part (first three rows) of Table 4.3 shows the results on ManyBugs benchmark and the second part gives the

Table 4.3: Evaluation results of ExtractFix. The first part (the first three rows) show the results on ManyBugs benchmark, and the second part present the results on our benchmark.

Application	Defects	CFC	FL (T1/T3)	Patches	Correct Patches	Avg. Time (m)
Libtiff*	5	3	2 / 3	3	2	4.32
Lighttd	3	2	1 / 2	2	1	7.50
Php	18	14	6 / 10	14	9	11.11
Libtiff*	11	9	7 / 8	9	6	5.64
Binutils	2	2	1 / 1	2	1	26.28
Libxml	5	4	3 / 3	4	2	13.80
Libjpeg	4	3	1 / 2	3	2	12.01
FFmpeg	2	2	1 / 1	2	2	8.23
Jasper	2	2	1 / 1	2	1	1.07
Coreutil	4	2	1 / 2	2	2	5.17
Total	56	43	24 / 33	43	28	9.46

* Both Manybugs and our benchmark include the Libtiff program, but the defects in different benchmarks do not overlap.

results on our own constructed subject. The effectiveness of ExtractFix is shown in columns 3-5, where *CFC* shows the number of correctly generated crash-free constraints in each application. Column `FL` represents fix localization results in a form of (*T1*/*T3*), where `T1` is the number of bugs whose correct fix location is ranked first, and *T3* is the number of bugs whose correct fix location is ranked in the top three candidates. `Patches` shows the number of fixed programs that pass the (single) failure-inducing test, while column `Correct Patches` are the number of generated patches that are semantically equivalent to developer patches. The detailed evaluation result of each defect can be found in Table 4.4 (ManyBugs) and Table 4.5 (our benchmark).

Crash-free constraint extraction. Out of 56 vulnerabilities, ExtractFix can successfully extract correct constraints for 43 defects, and all of them are correct according to our manual investigation. The results show that our constraint extraction can effectively extract crash-free-constraints, especially for integer overflow, divide-by-zero and developer assertions. We cannot extract correct constraint for some buffer overflow vulnerabilities and null pointer dereferences because the debugging information is ambiguous when symbolizing the condition enforced by sanitizers (the limitation of our prototype).

Table 4.4: Patches generated by ExtractFix on ManyBugs Benchmark. Column *Sanitizer* represents the sanitizer used by each defects, where *APISan* is a sanitizer implemented by ourselves to detect violation of API specification, e.g. the destination and source parameters should not overlap in *memcpy*. Column *Template* shows the template id (defined in Table 4.1) used by each vulnerability, while column *CFC* represents whether we can extract correct crash-free constraints. Column *FL* is the fault localization results, where L-N represents that we need to try *N* fix location candidates until find the correct one. Column *Patched* shows whether we can generate patches to pass the given tests. Column *Correct?* present the correctness of generated patches, where *Syn Equiv.* and *Sem Equiv.* means the generated patch is syntactically and semantically equivalent to developer patch and *Plausible* mean the generated patch pass the failing test but semantically incorrect. *Distance* presents the distance between fix location and crash location.

Subject	ID	Type	Sanitizer	Template	CFC	FL	Patched	Correct?	Distance	Time(m)
Libtiff	207c78a	ND	UBSan	T_5	✗	-	✗	—	—	—
	0a36d7f	BO	Lowfat	T_2	✓	L-1	✓	Sem Equiv.	4	3.44
	ee65c74	IO	UBSan	T_3	✓	L-3	✓	Plausible	10	5.66
	865f7b2	BO	Lowfat	T_2	✗	-	✗	—	—	—
	565eaa2	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	2	3.86
Lighttd	1914	BU	Lowfat	T_2	✗	-	✗	—	—	—
	2662	AS	Assert	T_1	✓	L-3	✓	Sem Equiv.	9	8.09
	2786	BO	Lowfat	T_2	✓	L-1	✓	Plausible	7	6.91
Php	5bb0a44e06	ND	UBSan	T_5	✓	L-4	✓	Plausible	10	16.23
	426f31e790	AA	APISan	T_4	✓	L-1	✓	Syn Equiv.	2	14.31
	2a6968e43a	BO	Lowfat	T_2	✓	L-1	✓	Sem Equiv.	2	12.09
	8deb11c0c3	ND	UBSan	T_5	✓	L-1	✓	Plausible	1	10.08
	7f2937223d	ND	UBSan	T_5	✗	-	✗	—	—	—
	2adf58cfcf	ND	UBSan	T_5	✓	L-2	✓	Syn Equiv.	5	9.89
	3acdca4703	ND	UBSan	T_5	✓	L-1	✓	Syn Equiv.	5	9.68
	c2fe893985	ND	UBSan	T_5	✗	-	✗	—	—	—
	93f65cdeac	ND	UBSan	T_5	✗	-	✗	—	—	—
	8d520d6296	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	1	7.89
	cacf363957	AA	APISan	T_4	✓	F	✓	Plausible	2	8.96
	c1e510aea8	ND	UBSan	T_5	✓	L-2	✓	Sem Equiv.	4	10.23
	f330c8ab4e	ND	UBSan	T_5	✓	L-5	✓	Sem Equiv.	32	10.24
	1d6c98a136	ND	UBSan	T_5	✓	F	✓	Plausible	2	30.02
	acaf9c5227	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	7	5.89
	032bbc3164	BO	Lowfat	T_2	✓	L-2	✓	Sem Equiv.	47	4.30
	1923ecfe25	AA	APISan	T_4	✗	-	✗	—	—	—
	cfa9c90b20	ND	UBSan	T_5	✓	L-1	✓	Plausible	1	5.66
Total	26	—	—	–	19	—	19	12	(avg) 8.1	(avg) 9.6

BO: buffer overflow; *BU*: buffer underflow; *IO*: integer overflow; *DZ*: divide-by-zero;
AA: API assert; *ND*: null pointer dereference; *AS*: developer assertion;

Fix localization. For the cases that we can extract correct constraints, we further evaluate the effectiveness of our fix localization. Out of 43 vulnerabilities, the correct fix locations of 24 defects are exactly the first candidate T1 recommended by our fix localization algorithm. The correct fix locations of 33 defects are correctly localized by looking into the top three candidates (T3).

Patch generation. Once constraints are correctly extracted and fix location

Table 4.5: Patches generated by ExtractFix. Column *Sanitizer* represents the sanitizer used by each defects, where *APISan* is a sanitizer implemented by ourselves to detect violation of API specification, e.g. the destination and source parameters should not overlap in *memcpy*. Column *Template* shows the template id (defined in Table 4.1) used by each vulnerability, while column *CFC* represents whether we can extract correct crash-free constraints. Column *FL* is the fault localization results, where L-N represents that we need to try *N* fix location candidates until find the correct one. Column *Patched* shows whether we can generate patches to pass the given tests. Column *Correct?* present the correctness of generated patches, where *Syn Equiv.* and *Sem Equiv.* means the generated patch is syntactically and semantically equivalent to developer patch and *Plausible* mean the generated patch pass the failing test but semantically incorrect. *Distance* presents the distance between fix location and crash location.

Subject	Vulnerability ID	Type	Sanitizer	Template	CFC	FL	Patched	Correct?	Distance	Time(m)
Libtiff	CVE-2016-5321	BO	Lowfat	T_2	✓	L-1	✓	Syn Equiv.	2	1.68
	CVE-2014-8128	BO	Lowfat	T_2	✓	L-1	✓	Sem Equiv.	5	2.40
	CVE-2016-5314	BO	Lowfat	T_2	✗	-	✗	—	—	—
	Bugzilla 2633	BO	Lowfat	T_2	✓	L-5	✓	Plausible	12	4.03
	CVE-2016-10094	BO	Lowfat	T_2	✓	L-2	✓	Plausible	2	1.87
	CVE-2016-3186	AA	APISan	T_4	✓	L-1	✓	Syn Equiv.	2	32
	CVE-2017-7601	IO	UBSan	T_3	✓	L-1	✓	Plausible	3	2.38
	CVE-2016-9273	BO	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2016-3623	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	2	2.05
	CVE-2017-7595	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	2	2.20
	Bugzilla 2611	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	1	2.13
Binutils	CVE-2018-10372	BO	Lowfat	T_2	✓	F	✓	Plausible	2	16.57
	CVE-2017-15025	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	2	36.00
Libxml2	CVE-2016-1834	IO	UBSan	T_3	✓	F	✓	Plausible	12	5.97
	CVE-2016-1839	BU	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2016-1838	BO	Lowfat	T_2	✓	L-1	✓	Plausible	3	4.12
	CVE-2012-5134	BU	Lowfat	T_2	✓	L-1	✓	Syn Equiv.	2	40.83
	CVE-2017-5969	ND	UBSan	T_5	✓	L-1	✓	Syn Equiv.	2	4.30
Libjpeg	CVE-2018-14498	BO	Lowfat	T_2	✓	L-10	✓	Plausible	3	1.22
	CVE-2018-19664	BO	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2017-15232	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	2	1.37
	CVE-2012-2806	BO	Lowfat	T_2	✓	L-3	✓	Sem Equiv.	10	33.26
FFmpeg	CVE-2017-9992	BO	Lowfat	T_2	✓	L-4	✓	Sem Equiv.	7	9.27
	Bugzilla-1404	IO	UBSan	T_3	✓	L-1	✓	Sem Equiv.	3	7.20
Jasper	CVE-2016-8691	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	5	1.08
	CVE-2016-9387	IO	UBSan	T_3	✓	F	✓	Plausible	5	1.05
Coreutil	Bugzilla-26545	AA	APISan	T_4	✓	L-3	✓	Syn Equiv.	4	6.03
	Bugzilla-25003	AA	APISan	T_4	✓	L-1	✓	Syn Equiv.	2	4.30
	GNUBug-25023	BO	Lowfat	T_2	✗	-	✗	—	—	—
	GNUBug-19784	BO	Lowfat	T_2	✗	-	✗	—	—	—
Total	30	—	—	—	24	—	24	16	(avg)4.0	(avg)9.3

BO: buffer overflow; *BU*: buffer underflow; *IO*: integer overflow; *DZ*: divide-by-zero;
AA: API assert; *ND*: null pointer dereference

candidates are determined, ExtractFix then generates patches via constraint propagation and program synthesis. Out of 56 vulnerabilities, ExtractFix can generate 43 patches. Those patches fix the bug by changing conditions, modifying the right-value of assignment or inserting an if-guard checker. For instance, to fix

Table 4.6: The number of patches and correct patches generated by Prophet, Angelix, Fix2Fit and ExtractFix (EF). The first part (the first three rows) shows the results on ManyBugs benchmark, and the second part presents the results on our benchmark. In each subject, the tool that produces the most patches and the most correct patches is marked in bold.

Program	#Vul	Total Patches				Correct Patches			
		Prophet	Angelix	Fix2Fit	EF	Prophet	Angelix	Fix2Fit	EF
Libtiff	5	2	**3**	**3**	**3**	1	1	1	**2**
Lighttd	3	**2**	**2**	**2**	**2**	0	0	0	**1**
Php	18	10	7	9	**14**	6	4	6	**9**
Libtiff	11	7	7	7	**9**	1	0	1	**6**
Binutils	2	-	-	1	**2**	-	-	0	**1**
Libxml2	5	3	0	**4**	**4**	0	0	1	**2**
Libjpeg	4	**3**	-	-	**3**	1	-	-	**2**
FFmpeg	2	-	-	**2**	**2**	-	-	1	**2**
Jasper	2	**2**	**2**	**2**	**2**	0	0	0	**1**
Coreutil	4	2	-	**3**	2	0	-	1	**2**
Total	56	31	21	33	**43**	9	5	11	**28**

the *Libtiff* buffer overflow of CVE-2014-8128, developers add an if-checker at line 571 to break the *while*-loop when *nrows* is equal to 256:

```
571 + if (nrows == 256) break;
```

Instead, ExtractFix fixes the bug by modifying the exit condition of *while*-loop, which is semantically equivalent to the developer patch:

```
567 - while (err >= limit)
567 + while (err >= limit && nrows < 256)
```

With this patch, it is guaranteed that the vulnerability cannot be triggered again. In our benchmark, once a correct constraint is generated, ExtractFix can always generate a patch.

Multi-line fix To fix the *Libjpeg* buffer overflow vulnerability of CVE-2012-2806, ExtractFix generates multiple-line fixes by changing two *for*-loop conditions.

Comparison with state-of-the-art We then compare the repairability of ExtractFix with Prophet, Angelix and Fix2Fit. We cannot run Angelix on some applications because the libraries (e.g. clang 2.9) used by Angelix no longer support the new versions of those applications. We did not run Fix2Fit on Libjpeg since it does not

support the compilation using cmake. Prophet fails to build Binutils and FFmpeg. The columns 3-6 of Table 4.6 represent the number of patches generated by Prophet, Angelix, Fix2Fit and ExtractFix, respectively. Compared with Prophet and Angelix, ExtractFix generates the same or more patches for all the applications. Compared with Fix2Fit, ExtractFix generates more patches on Php, Libtiff, and Binutils, but less on Coreutils. This is because Fix2Fit generates plausible patches by efficiently searching from a large patch space and then uses fuzzing to rule out overfitted patches. In fact, our comparison with Fix2Fit is conservative in favor of Fix2Fit, since Fix2Fit's fuzzing campaigns have an 8-hour timeout, while our program analysis based technique has a timeout of 1 hour (30 minutes for symbolic execution and 30 minutes for program synthesis). Even then, ExtractFix generates more plausible patches than Fix2Fit. More importantly, as we will see later, the patches generated by ExtractFix are of significantly higher quality than the patches from Fix2Fit.

Out of 56 vulnerabilities, ExtractFix extracts 43 correct constraints and generates 43 patches. ExtractFix generates more patches than Prophet, Angelix and Fix2Fit.

4.5.2.2 Can ExtractFix alleviate the overfitting problem?

The generated patch can handle the bug-triggering exploit, but it may overfit to the given exploit. To evaluate patch correctness, we take the developer patch as criteria and examine the patch correctness by manually analyzing the developer patch. For each generated patch by ExtractFix, we check its syntactic and semantic equivalence with the developer patch by manually examining if the patch changes the program behavior in the same way as the developer patch.

In Table 4.3, column *Correct Patch* gives the number or patches that are syntactically or semantically equivalent to developer patches. Out of the 43 patches, 28 patches are syntactically or semantically equivalent to developer patches, while 15 of them are plausible patches. We mark a patch as *Plausible* if it partially fixes the vulnerability or changes program behavior differently compared to the developer patch. Plausible patches exist because (1) the CFC' could be incomplete

since backward propagation misses some paths between fix and crash location (e.g. paths inside *for*, *while* loop) due to the practical limitation of symbolic execution (2) ExtractFix knows how to avoid triggering the vulnerability, but has narrow knowledge about the intended program behavior from developers. For instance, an integer overflow CVE-2017-7601 occurs when performing shift operation ($1L << bitssample$) with $bitssample >= 63$ (maximal positive signed long integer is $2^{63}-1$). To fix this vulnerability, developers insert an if-checker (*if*($bitssample > 16$) *return 0*) before the crash line. With the guidance of crash free constraint $bitssample < 63$, ExtractFix fix the bug by inserting *if*($bitssample >= 63$) *return 0*. The generated patch completely fixes the integer overflow, but may unintentionally modify the other program behaviors. While ExtractFix is designed to alleviate overfitting by completely fixing vulnerabilities, it may still change the program behavior in an unintended way.

We compare ExtractFix with Prophet, Angelix, and Fix2Fit for patch quality. The evaluation results are shown in Table 4.6, where columns 7-10 represent the number of correct patches generated by Prophet, Angelix, Fix2Fit, and ExtractFix, respectively. The correctness of patches is examined by manually comparing with the developer patches to check their semantic equivalence. The test suite provided to repair tools is composed of the exploit and all available developer tests. Prophet, Angelix, and Fix2Fit are test-driven program repair tools, so the quality of patches generated by them highly depends on the quality of the test suite. For the defects from ManyBugs, test-driven program repair tools have a higher chance to generate correct patches since there are more available tests. On average, there are around 2.9k available tests for each defect from ManyBugs benchmarks (the first part of Table 4.6).[4] All the test-driven program approaches generate a number of correct patches. Specifically, on the 26 defects, Prophet, Angelix, and Fix2Fit generate 7, 5, and 7 correct patches, respectively. Even then, ExtractFix generates much more (12) correct patches than all those approaches.

In our constructed benchmark (the second part of Table 4.6), the available tests are very limited, and only very few tests can cover the crash line. Therefore, the generated patches by these tools can easily overfit the given tests. Specifically, by

[4]Around half of the 2.9k tests are irrelevant and will not drive the program to the fix locations. Even then, there are still a considerable number of useful tests in each subject.

manually checking the top patches against developer patches, only two patches generated by Prophet are correct and all the patches from Angelix overfit the failing tests. Fix2Fit can filter out some overfitted patches by test case generation, but the quality of the patches is not high as found by our experiments. Out of the 20 patches generated by Fix2Fit, only four patches are correct, while others still overfit the given test suite. In contrast, ExtractFix generates as many as 16 correct patches.

> For bugs that are vulnerabilities supported by ExtractFix, ExtractFix outperforms Prophet, Angelix and Fix2Fit in generating patches that are both syntactically and semantically equivalent to developer patches.

4.5.2.3 How efficient is ExtractFix in generating patches?

Scalability is one of the most challenging problems of symbolic execution, hence semantic-based program repair. In our evaluation, we show that our approach can scale to real-world large applications, e.g. FFmpeg with 617K lines of codes. Meanwhile, the execution time to generate patches is given in Table 4.3. On average, we only need 9.46 minutes to generate a patch, with a maximum of 41 minutes. Our approach is efficient because (1) our symbolic execution is only performed on a small partial program. As shown in Table 4.3 and 4.5, the averaged distance between fix and crash location is around 6, with maximum of 47. (2) our second-order program synthesis takes into account the distance between patch candidates with original expression and first evaluates candidates that are close to the original expression.

> ExtractFix can scale to large programs, such as FFmpeg. On average, it takes 9.46 minutes to generate patches.

4.5.3 Threats to Validity

Internal Validity The main threat to internal validity is that ExtractFix performs backward propagation via symbolic execution which may miss some paths and result in incomplete constraint propagation. Fortunately, we only perform symbolic execution on a very small part of the program. What matters is that

the incompleteness doesn't seem to have a big impact on the effectiveness of the analysis. Another threat to internal validity is that we derive our *CFC* templates from frequently reported bugs and vulnerabilities, we note that our set of templates is not exhaustive. By extending *CFC* templates, ExtractFix can easily support fixing other kinds of bugs/vulnerabilities whose property violation is sanitizable and expressible as a simple formula. The last internal threat is that we perform a manual inspection of the experimental results which might be error-prone. To mitigate this, we have double-checked the generated patches.

External Validity The main threat to external validity is that our selection of subjects may not generalize to other programs. We cannot evaluate ExtractFix on the dataset used in [128, 44, 63], because FootPatch fixes resource/memory leak (C/C++) and null pointer dereference (Java), a large part of defects in ManyBugs are logic bugs, and the exploits and fixes of some datasets (exploits) used by SENX are not available. Instead, we evaluate ExtractFix on a set of real programs and real CVEs to show its usability. In the future, it may be worthwhile to evaluate our approaches on more relevant CVEs and bugs.

Chapter 5

Alleviate Overfitting Using Semi-Supervised Synthesis

Even though ExtractFix can provide guarantees, it is not scalable since it relies on symbolic execution. We propose another test case abstraction or generalization idea based on a semi-supervised approach. Inspired by semi-supervised learning which trains a model based on both labeled data and unlabelled data, we present a semi-supervised synthesis in this chapter. Different from traditional program synthesis, semi-supervised synthesis takes into account both user-provided examples (corresponds to labeled data) and additional inputs (corresponds to unlabelled data). The main insight is that the additional inputs can guide the program synthesis process by providing more inputs that should be manipulated by the synthesized program. We apply semi-supervised synthesis for program transformations.

5.1 Introduction

Integrated Development Environments (IDEs) and static analysis tools help developers edit their code by automating common classes of edits, such as boilerplate code edits (e.g., equality comparisons or constructors), code refactorings (e.g., rename class, extract method), and quick fixes (e.g., fix possible `NullReferenceException`). To automate these edits, tool builders implement *code transformations* that manipulate the Abstract Syntax Tree (AST) of the user's code to produce the desired code edit.

While traditional tools support a predefined catalog of transformations handcrafted by tool builders, in recent years, we have seen an emerging trend of tools and techniques that synthesize program transformations using examples of code

edits [81, 109, 82, 87, 7, 110]. For instance, GETAFIX [7] learns fixes for static analysis warnings using previous fixes as examples. It has been deployed at Facebook where it is used for the maintenance of Facebook apps. BLUEPENCIL [87] produces code edit suggestions to automate repetitive code edits, i.e., edits that follow the same structural pattern but that may involve different expressions. It synthesizes transformations on-the-fly based on the recent edits performed by the developer. BLUEPENCIL has been released in Microsoft Visual Studio 2019 [84] and is available as Visual Studio IntelliCode suggestions [85].

The main challenge of synthesizing an intended transformation program from examples lies in that the synthesized program should not only satisfies the given examples but also produces the correct edits on unseen inputs. Overfitted transformation program can lead to *false negatives*: the transformation does not produce an edit suggestion in a location that should be changed. False negatives increase the burden on developers, since it requires developers to either provide more examples or perform the edits themselves, reducing the number of automated edits. Moreover, it may cause developers to miss edits leading to bugs and inconsistencies in the code. Overfitted transformation program can also lead to *false positives*: the transformation produces an incorrect edit. While false negatives are usually related to transformations that are too specific, false positives are mostly related to transformations that are too general. Both false negatives and positives can reduce developers' confidence in the aforementioned systems, and thus, finding the correct generalization is crucial for the adoption of these systems.

Existing approaches have tried to handle the generalization problem in different ways. SYDIT [81] and LASE [82] can only generalize names of variables, methods and fields when learning a code transformation. The former only accepts one example and synthesizes the transformation using the most general generalization. The latter accepts multiple examples and synthesizes the transformation using the most specific generalization, which is also the approach adopted by REVISAR [110] and GETAFIX [7]. Using either the most specific or the most general generalization is usually undesirable, as they are likely to produce false negatives and false positives, respectively. REFAZER [109] learns a set of transformations consistent with the examples and stores them as a Version Space Algebra (VSA) [88]. It then uses a ranking system to rank the transformations and selects the one that is more likely

to be correct based on a set of predefined heuristics. However, despite the more sophisticated approach to generalization, in certain cases, ReFazer still requires up to six examples of a repetitive edit before producing edit suggestions [109].

All aforementioned techniques rely only on input-output examples of edits and background knowledge in the form of ranking schemes and heuristics to deal with the generalization problem. However, apart from these, an additional source of information could be the large number of *additional input trees* available in the remainder of the file and project the user is editing. Semi-supervised learning [148] is an approach to machine learning that combines a set of labeled input-output examples and unlabeled data (inputs) during training. It has recently become more popular and practical due to the variety of problems for which vast quantities of unlabeled data are available, e.g. text on websites, protein sequences, or images [149]. The fact that many additional inputs are available in source code inspires a natural question:

Is it possible to combine input-output examples with additional inputs to synthesize program transformations?

Our first key observation is that an additional input AST can help us disambiguate how to generalize the transformation by providing more examples of ASTs that should be manipulated by the transformation. Consider a simple change from `if (score < limit)` to `if (IsValid(score))`. With a single example, it is not clear whether we do the transformation only when the left-hand side of the comparison is `score`. However, if one says that the transformation should also apply to `if (GetScore(run) < limit)`, then we have one more example for the LHS expression, `GetScore(run)`, and we can use this example to refine our transformation—in this case, generalize it further. However, we still need to identify the locations in the source code (the additional inputs) where the transformation should apply. Our second key observation is that we can predict whether an arbitrary input should be an additional input by evaluating the quality of the transformation synthesized when using the new input. The quality is assessed using a user-driven or automated feedback system.

We propose a feedback-driven semi-supervised technique to synthesize program transformations. The proposed approach is based on our two key observations above.

Initially, our technique synthesizes a program transformation from input-output examples using ReFazer [109]. For the input-output example, it tracks which subtrees of the AST (corresponding to a sub-expression) were used to construct the output, and can potentially be generalized. We call these nodes *selected nodes*. As an example, consider again the change `if (score < limit)` to `if (IsValid(score))`. The expression `score` was used in the output—it is a selected node. Next, our technique iterates over candidate additional inputs to find more examples to refine the generalization. For each candidate input, it performs two main steps:

- First, our technique computes the *anti-unification* of the examples and the candidate additional input. Anti-unification is a generalization process which can identify corresponding subtrees among different input ASTs. For instance, it can identify that `score` in the example input corresponds to `GetScore(run)` in the candidate additional input `if (GetScore(run) < limit)`. Our anti-unification based generalization algorithm tries to compute a generalization where each selected node in the example input has a corresponding node in the candidate additional input. For example, if the candidate additional input was `if (UnrelatedCondition())`, then we can infer the correspondence between `(score < limit)` and `(UnrelatedCondition())`, and the subtree `score` itself has no corresponding subtree, which causes anti-unification to fail to find a generalization. If anti-unification fails, the candidate additional input is not compatible and we discard it. Otherwise, we generate a new example from the candidate additional input, and re-synthesize parts of the transformation while taking this example into consideration. In our running scenario, the new example is `if (GetScore(run) < limit)` $\mapsto$ `if (IsValid(GetScore(run)))`.

- Then, our technique uses a feedback system to further evaluate whether the current candidate input should be accepted. The feedback is provided by a *reward function* that can be composed of different components. It can take into consideration user-provided feedback, for example, if the transformation should apply to a particular input. Indicating such inputs is usually an easier task for the user than providing another input-output example. However, the feedback can also use automated components based on, for example, the similarity of the additional input to the example inputs. If the final reward score is above

a certain threshold, it accepts the additional input and synthesizes a new program transformation using the new example.

We implemented our technique for the domain of C# program transformations. It uses the implementation of ReFazer available in the PROSE SDK[1]. Further, we augmented the BluePencil algorithm [87] with our approach to synthesize on-the-fly transformations. BluePencil provides a *modeless interface* where developers do not need to enter a special mode to provide examples, but instead, they are inferred from the history of changes to a particular file.

With these components, we implemented three applications that use feedback-driven semi-supervised synthesis:

- ReFazer*: *User-provided feedback about additional inputs.* This application allows developers to specify, as an additional input, a subtree where the transformation did not produce an edit (false negative). This implementation is motivated by the fact that when the transformation-learning system produces a false negative, it is easier for the developer to provide an additional input rather than a complete input-output example. On a benchmark of 12,642 test cases, we compared ReFazer* with the baseline (ReFazer). While the recall of ReFazer ranged from 26.71% (1 example provided) to 89.10% (3 examples provided), the recall of ReFazer* was at least 99.94% and its precision was at least 96.01% with just 1 example and 1 additional input provided. These results suggest that ReFazer* can synthesize suggestions with high precision at locations indicated by developers as false negatives.

- BluePencil$_{\mathsf{cur}}$: *Semi-automated feedback based on cursor position.* This feature uses the cursor position in the editor to indicate candidate additional inputs to semi-supervised synthesis. This feature is motivated by the fact that the developers may either not be aware that they can provide additional inputs (discoverability problem [87]), or may not want to break their workflow to provide additional inputs. The cursor position acts as a proxy for the user and indicates, implicitly, that the user wants to modify the current location. However, the cursor location is ambiguous. The subtree that the user wants to edit may be any of the subtrees

[1] `https://www.microsoft.com/en-us/research/group/prose/`

that are present at the cursor location, i.e., the lowest leaf node at the cursor location all the way to the root of the AST. The tool relies on feedback from a reward function (Section 5.4.2) to accept additional inputs. We compared this reward function with two alternative reward functions: (i) *no validation*, where semi-supervised synthesis accepts any additional inputs; (ii) and *clone detection* where semi-supervised synthesis accepts inputs based on their similarities with the inputs in the input-output examples. Our results show that while "no validation" and "clone detection" lead to high false positives and negatives, respectively, our reward function produces only 11 false positives and 14 false negatives on 243,682 tested additional inputs. We also evaluated the effectiveness of BluePencil$_{\mathsf{cur}}$ in generating correct suggestions at the cursor location. Amongst 291 scenarios, BluePencil$_{\mathsf{cur}}$ only generates one false positive and three false negatives.

- BluePencil$_{\mathsf{auto}}$: *Fully-automated feedback based on all inputs in the source code.* This feature uses all the nodes available in the source code as input to semi-supervised synthesis. It is relevant in the settings where user feedback is not available. For example, (a) when the developer themselves may not be aware of all locations that must be changed, or (b) when the developer may want to apply the edits in bulk, instead of inspecting each one for correctness. We evaluated how often BluePencil$_{\mathsf{auto}}$ can save developers from indicating the additional inputs. To do so, we simulated a developer performing 350 repetitive edits with BluePencil$_{\mathsf{cur}}$ and BluePencil$_{\mathsf{auto}}$ enabled or just BluePencil enabled. In our experiment, BluePencil$_{\mathsf{auto}}$ decreased the number of times the developer would have to indicate the input by 30%. When compared to BluePencil, our results show that BluePencil$_{\mathsf{cur}}$ and BluePencil$_{\mathsf{auto}}$ automated 263 edits while BluePencil automated only 159.

Remark 5.1.1 *In this chapter, the term* semi-supervised *is used in a subtly different manner than in the traditional machine learning context. In both settings, additional unlabelled inputs are used to aid learning. However, in machine learning, the additional unlabelled inputs are used to understand the structure and distribution of the input space. On the other hand, in our setting, additional inputs are used to generate new input-output examples along the lines of existing labeled examples, using the structure of individual additional input trees. In other words, semi-supervised*

(a) Two repetitive edits. Both edits update invocations to method `ResolveDependency` but one of the arguments is different. Given these two edits, IntelliCode synthesizes a transformation to automate similar edits.

```
-  repository.ResolveDependency(dependency1, null, false, false, Lowest);
+  DependencyResolverUtility.ResolveDependency(repository, dependency1, null,
     false, false, Lowest);

-  repository.ResolveDependency(dependency2, null, false, false, Lowest);
+  DependencyResolverUtility.ResolveDependency(repository, dependency2, null,
     false, false, Lowest);
```

(b) IntelliCode correctly produces suggestions at these locations based on the previous edits. The first argument is the only difference between these locations, similar to the examples.

```
  repository.ResolveDependency(dependency3, null, false, false, Lowest);
  repository.ResolveDependency(dependency4, null, false, false, Lowest);
```

(c) IntelliCode fails to produce suggestions to these locations (false negative). Note that there are more elements that are different in these locations compared to the locations in the examples.

```
  repository.ResolveDependency(dependency1, null, false, false, Highest);
  repository.ResolveDependency(dependency2, null, false, false, Highest);
  Marker.ResolveDependency(dependency, null, AllowPrereleaseVersions, false,
      Highest);
```

(d) While this location shares the same structure as the previous ones, the transformation should not produce an edit here.

```
- s.GetUpdates(IsAny<IEnumerable<IPackage>>(), false, false,
+ DependencyResolverUtility.GetUpdates(s, IsAny<IEnumerable<IPackage>>(),
    false, false, IsAny<IEnumerable<FrameworkName> >(),
    IsAny<IEnumerable<IVersionSpec> >())
```

Figure 5.1: A scenario with two repetitive edits (input-output examples), additional inputs, and a false positive. All inputs share the same structure (a method invocation with 5 arguments).

machine learning exploits the structure of the input space, while we use the structure of individual inputs.

5.2 Motivating Example

We start by illustrating the challenges of synthesizing code transformations from input-output examples. Consider the scenario shown in Figure 5.1.

A C# developer working on NuGet[2] codebase refactored the `ResolveDependency`

[2]Nuget is a package manager for .NET

Figure 5.2: BluePencil$_{cur}$ implemented as a Visual Studio extension. The developer clicks on a line to manually edit the code where the PBE system produced a false negative. BluePencil$_{cur}$ uses feedback-driven program-synthesis to synthesize a transformation that is general enough to be applied to this location. The edit generated by the transformation is shown as an auto-completion suggestion.

method to make it static, then moved it to static class `DependencyResolveUtility`. As a result, the developer must update all invocations of this method to match its new signature. Figure 5.1a shows two call sites where the developer has manually updated the invocation to match this signature. Figures 5.1b and 5.1c show additional locations that will require a similar modification: note that they share the same general structure but contain dissimilar subexpressions. Manually performing such repetitive edits is tedious, error-prone, and time-consuming. Unfortunately, developer tools such as the Visual Studio IDE [84] and ReSharper [47] do not include built-in transformations or refactorings to automate these edits.

However, a recently introduced Visual Studio feature based on BluePencil [87], called IntelliCode suggestions (IntelliCode for brevity), can learn to automate these edits after watching the developer perform a handful of edits. Specifically, after watching edits to the two locations shown in Figure 5.1a, IntelliCode learns a transformation and suggests automated edits to the locations shown in Figure 5.1b.

With only these two examples, however, IntelliCode is not yet able to produce suggestions for the locations shown in Figure 5.1c. These are *false negatives*. This is because the inputs in the examples provided so far differed only in their first method argument: `dependency1` and `dependency2`, respectively. As a result, IntelliCode synthesizes a transformation that generalizes across variation in the first argument, but not the others. While sufficient to suggest edits for the locations in Figure 5.1b, this transformation is not sufficiently general to apply to the locations shown in

Figure 5.1c, which contain additional variation in the call target, third argument, and fifth argument (`Marker`, `AllowPrereleaseVersions`, and `Highest`, respectively).

To address this situation, the developer performs another manual edit at the first location in Figure 5.1c. IntelliCode consumes this edit as a new example and synthesizes a new transformation to generalize across variation in both the first and fifth arguments: IntelliCode has disambiguated the developer's intent because the new example contains a different variable (`Highest` rather than `Lowest`) in the final argument. At this point, IntelliCode is now able to produce correct suggestions for all locations that differ only in their first or last argument. Unfortunately, despite having seen three input-output examples, it still fails to produce suggestions for the last location in Figure 5.1c.

In general, false negatives like those described stem from insufficiently general transformations–they overfit to the given examples. They not only reduce the applicability of the tool but also frustrate developers, who naturally expect an edit suggestion to automate their task after having already supplied several examples. The line between *too specific* and *too general* can be thin, though. In this scenario, the desired transformation should produce edits on invocations of the instance method `ResolveDependency` using 5 arguments. If we generalize the name of the method to any method, it will lead to false positives. For instance, it would produce the edit shown in Figure 5.1d.

Our Solution. We now illustrate how a system based on semi-supervised synthesis can help alleviate this problem. BluePencil$_{\mathsf{cur}}$ uses the cursor position in the editor to indicate candidate additional inputs to our semi-supervised synthesis technique. Consider the first false negative shown in Figure 5.1c. As soon as the developer places the cursor in the location related to the false negative, BluePencil$_{\mathsf{cur}}$ uses our semi-supervised feedback synthesis technique to improve the transformation. The new transformation produces an *auto-completion* suggestion for the current location (see Figure 5.2). We provide details of our technique and its applications in Sections 5.4 and 5.5, resp. In the next section, we formalize the problem of feedback-driven semi-supervised synthesis.

5.3 The Semi-Supervised Synthesis Problem

We first formalize the semi-supervised synthesis problem and then discuss the feedback-driven semi-supervised synthesis problem.

Abstract Syntax Trees. Let $\mathbb{T}$ denote the set of all abstract syntax trees (AST). We use the notation t to denote a single AST in $\mathbb{T}$, and use the notation $\mathsf{SubTrees}(t) \subseteq \mathbb{T}$ to denote the set of all subtrees in t. Each node in the AST consists of a string label representing the node type (e.g., Identifier, MethodDeclaration, InvokeExpression, etc), set of attributes (e.g., text value of leaf nodes, etc) and a list of children ASTs.

Edit Programs. An *edit program*[3] $\mathsf{P} : \mathbb{T} \nrightarrow \mathbb{T}$ is a partial function[4] that maps ASTs to ASTs. In this chapter, we assume that each edit program P is a pair $(\mathsf{P}_{\mathsf{guard}}, \mathsf{P}_{\mathsf{trans}})$ of two parts: (a) a *guard* $\mathsf{P}_{\mathsf{guard}} : \mathbb{T} \to \mathbb{B}$, and (b) a *transformer* $\mathsf{P}_{\mathsf{trans}} : \mathbb{T} \nrightarrow \mathbb{T}$. We have that $\mathsf{P}(t) = \mathsf{P}_{\mathsf{trans}}(t)$ when $\mathsf{P}_{\mathsf{guard}}(t)$ is true, and $\mathsf{P}(t) = \perp$ otherwise.

Example 5.3.1 *Consider the two edits shown in Figure 5.1a. For each edit, the following edit program maps the subtree before the change to the subtree after the change.*

$$
\begin{aligned}
\mathsf{P}_{\mathsf{guard}} = \;& \text{Input matches } \mathsf{X}_1.\mathsf{X}_2(\mathsf{X}_3, \mathsf{X}_4, \mathsf{X}_5, \mathsf{X}_6, \mathsf{X}_7) \text{ where} \\
& \quad \mid \mathsf{X}_1.\mathsf{label} = \mathsf{Identifier} \wedge \mathsf{X}_1.\mathsf{Attributes.TextValue} = \texttt{repository} \\
& \quad \mid \mathsf{X}_2.\mathsf{label} = \mathsf{Identifier} \wedge \mathsf{X}_2.\mathsf{Attributes.TextValue} = \texttt{ResolveDependency} \\
& \quad \mid \mathsf{X}_3.\mathsf{label} \wedge \ldots \\
\mathsf{P}_{\mathsf{trans}} = \;& \text{return } \texttt{DependencyResolveUtility}.\mathsf{X}_2(\mathsf{X}_1, \mathsf{X}_3, \mathsf{X}_4, \mathsf{X}_5, \mathsf{X}_6, \mathsf{X}_7)
\end{aligned}
$$

ReFazer learns this program initially in Section 5.2 (with just 2 examples). This program is written in terms of templates with each X_i representing a hole. In Section 2.2.2, we present a domain-specific language to express such programs.

The Semi-supervised Synthesis Problem. As explained in Section 5.2, the semi-supervised synthesis problem is the core piece among the techniques in this work. Semi-supervised synthesis allows a user or an environment to finely control the level of generalization used by the synthesizer. The formal definition of the problem is as follows. Given (a) a set of input-output examples $\mathsf{Examples} = \{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_k \mapsto \mathsf{o}_k\}$,

[3] We also refer to edit programs more generally as *transformations*.

[4] We consistently use $\nrightarrow$ to denote partial functions.

(b) a set of additional positive inputs $\mathsf{PI} = \{\mathsf{pi}_0, \ldots, \mathsf{pi}_n\}$, and (c) a set of additional negative inputs $\mathsf{NI} = \{\mathsf{ni}_0, \ldots, \mathsf{ni}_m\}$, the *semi-supervised synthesis problem* is to produce a program P such that (a) $\forall 0 \leq j \leq k.\mathsf{P}(\mathsf{i}_j) = \mathsf{o}_j$, (b) $\forall 0 \leq j \leq n.\mathsf{P}(\mathsf{pi}_j) \neq \bot$, and (c) $\forall 0 \leq j \leq m.\mathsf{P}(\mathsf{ni}_j) = \bot$. Intuitively, the problem asks for a program that is consistent with the provided examples, produces outputs on all additional positive inputs, and does not produce an output on any additional negative inputs. The over-generalization and under-generalization problem can be addressed by providing more additional negative and positive examples, respectively.

Feedback-Driven Semi-supervised Synthesis Problem. The semi-supervised synthesis problem assumes access to positive and negative additional inputs, but how do we find (more of) them to help refine the synthesized program? We use feedback from either the user or the environment to discover these additional inputs. In this setting, the synthesizer is provided with the following components: (a) A finite *pool of inputs* $\mathsf{InputPool} \subseteq \mathbb{T}$. We assume that all example inputs and additional (positive or negative) inputs are drawn from the input pool $\mathsf{InputPool}$. In practice, the input pool is usually the set of all subtrees of the AST representing a source file. (b) A *reward function* $\mathsf{Rew} : \mathsf{InputPool} \not\to [-\infty, \infty]$ that acts as a feedback mechanism. A high and a low reward for an $\mathsf{i} \in \mathsf{InputPool}$ indicates whether the synthesized program should be applicable to i or not, respectively. For exposition purposes, we separate the reward function into the *user provided* Rew_U and *environment provided* Rew_E reward functions with Rew being a combination of the two. In Section 5.4.2, we define *feedback oracles* which take as input the state of the feedback loop (i.e., examples, positive and negative inputs, synthesized program) and return a reward function. While we could merge the notion of feedback oracle and reward function, with reward function taking additional inputs mentioned, this separation allows for easier notation.

The rewards are generated from a number of factors including (a) if the user manually indicates whether an input from the input pool should be positively or negatively marked, (b) whether applying a produced edit leaves the source code document in a compilable state, and (c) whether the produced edit for an input is similar to or different from the given examples.

This workflow proceeds in multiple rounds of interaction. In the n^{th} iteration of

the workflow,

- The synthesizer, using the examples and the reward function Rew_{n-1}, produces a program P_n that is consistent with the examples $\mathsf{Examples}$ and the positive (and negative) additional inputs deduced from Rew_{n-1}.
- Optionally, the user adds new examples to the set of $\mathsf{Examples}$ to produce $\mathsf{Examples}_n$.
- The user and the environment in conjunction produce the rewards $\mathsf{Rew}_n : \mathsf{SubTrees}(t_n) \not\to [-\infty, \infty]$ to provide feedback on how P_n is to be refined in the next iteration to produce P_{n+1}.

This workflow is a continuous interaction between the environment and the user on one side, and the synthesizer on the other. This continuous interaction using rewards is reminiscent of a reinforcement learning scenario. However, in our setting, the user and the environment cannot be modeled as a Markov decision process, and the state space is non-continuous infinite, making standard reinforcement learning techniques not applicable.

Due to the user-in-the-loop nature of the feedback-driven semi-supervised synthesis workflow, it is hard to define an explicit correctness condition for the problem. The real optimality criterion for the synthesized program is *how well does the synthesized program match user intent?* This criterion is hard to capture formally in practice, mainly because users may not be willing or not be able to manually analyze the synthesized program. Further, depending on the scenario, the same program may either be correct or incorrect. For example, in the case from Section 5.2, in a slightly different scenario, it is quite possible that the under-generalized transformer generated initially is the intended transformation. It is impossible to guess without semantic knowledge about the domain of the source code, which we are consciously keeping out-of-scope here.

However, we do have a *quiescence condition* on the environment and the synthesizer combined: when the user-dependent feedback stops changing (i.e., Rew_U is fixed), the synthesized program should converge to a fixed one. Note that quiescence may be impossible under the situation where the user keeps adding more feedback or positive and negative examples. Due to the lack of strict correctness conditions, to

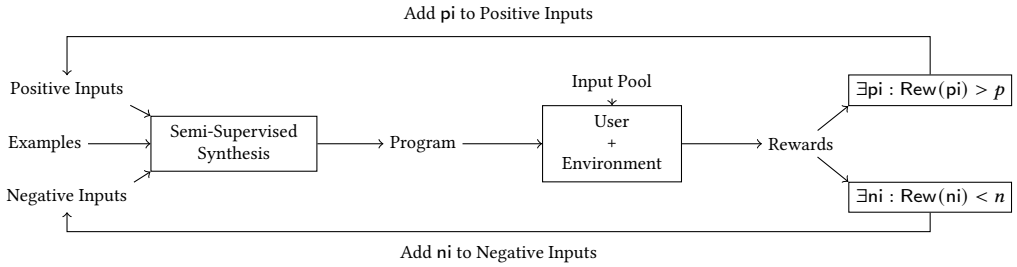


Figure 5.3: Solution for the feedback-driven semi-supervised problem

ensure the quality of the programs and edits produced, we experimentally validate the techniques with a comprehensive evaluation (Section 5.6).

5.4 Feedback-driven Semi-Supervised Synthesis

We present our technique to address the feedback-driven semi-supervised synthesis problem. This solution approach is depicted in Figure 5.3 and works as follows:

- In each round, the feedback-driven problem with real number feedback is converted into an instance of the semi-supervised synthesis problem. We achieve this reduction by choosing thresholds p and n, with $\mathsf{PI} = \{\mathsf{i} \in \mathsf{InputPool} \mid \mathsf{Rew}_{n-1}(\mathsf{i}) > p\}$ and $\mathsf{NI} = \{\mathsf{i} \in \mathsf{InputPool} \mid \mathsf{Rew}_{n-1}(\mathsf{i}) < n\}$.
- The semi-supervised synthesis is solved using a standard (not semi-supervised) program synthesizer. To ensure that the synthesized program produces outputs on the additional positive inputs, we generate new examples by associating each additional positive input pi with an output po. This output is produced using a given example $\mathsf{i} \mapsto \mathsf{o}$, and a combination of provenance analysis and anti-unification. Informally, we first associate each subtree s' of pi with an equivalent subtree s of i. Then, in o we replace each subtree generated from a subtree s of the input i, with a new subtree that is generated in a similar way but with s replaced by s'.

5.4.1 Semi-Supervised Synthesis

Algorithm 5 depicts a procedure for the semi-supervised synthesis problem. In the procedure, we use $\mathsf{ReFazer}_{\mathsf{guard}}$ and $\mathsf{ReFazer}_{\mathsf{trans}}$ as oracles. Oracle $\mathsf{ReFazer}_{\mathsf{guard}}$

Algorithm 5: Semi-supervised synthesis

Input: Input-output examples $\mathsf{Examples} = \{\mathsf{i}_0 \mapsto \mathsf{o}_0, \ldots, \mathsf{i}_k \mapsto \mathsf{o}_k\}$
Input: Additional positive inputs $\mathsf{PI} = \{\mathsf{pi}_0, \ldots, \mathsf{pi}_n\}$
Input: Additional negative inputs $\mathsf{NI} = \{\mathsf{ni}_0, \ldots, \mathsf{ni}_m\}$
Output: Program P

```
1  Inputs ← {i | (i ↦ o) ∈ Examples}
2  P_guard ← ReFazer_guard(Inputs ∪ PI, NI)
3  P_trans ← TransSynth(Examples, PI)
4  if P_guard = ⊥ ∨ P_trans = ⊥ then
5  |  return ⊥
6  end
7  return (P_guard, P_trans)
8
9  Function TransSynth(Examples, PI) is
10 |  P_trans ← ReFazer_trans(Examples)
11 |  π ← Provenance(i_0 ↦ o_0, P_trans)
12 |  (τ, ⟨σ_0, ..., σ_k, σ'_0, ..., σ'_n⟩) ← ⋈_π{i_0, ..., i_k, pi_0, ..., pi_n}
13 |  if ⊥ ∈ (σ_0, ..., σ_k, σ'_0, ..., σ'_n) then
14 |  |  return ⊥
15 |  end
16 |  AdditionalExamples ← {pi_j → Evaluate*(P_trans, pi, i) | pi_j ∈ PI}
17 |  return ReFazer_trans(Examples ∪ AdditionalExamples)
18 end
```

takes positive inputs and negative inputs, and produces a guard that is true on the former and false on the latter. Oracle $\mathsf{ReFazer}_{\mathsf{trans}}$ takes a set of examples and produces a transformer consistent with them.

The guard synthesis component of the algorithm (line 2) falls back to $\mathsf{ReFazer}_{\mathsf{guard}}$. However, transformer synthesis is significantly more involved. First, using only $\mathsf{Examples}$, we synthesize a transformer program that is consistent with each example (line 10). Using this program, we extract *provenance information* (line 11) on what fragments of the example outputs are dependant on what fragments of the example inputs, and what sub-programs are used to transform the input fragments to the output fragments. Then, we use *anti-unification* (line 12) to determine which fragments of the example inputs are associated with which fragments of the additional positive inputs. Using the provenance and anti-unification data, we can now compute a candidate output for each additional positive input (line 16). Finally, we synthesize a transformer program from the original examples and the new examples obtained by associating each additional positive input with its candidate

output. We explain these steps in detail below.

Provenance. The first step of transformer synthesis computes *provenance* information for each example. The provenance information is computed for `select` operations. Given a transformer program $\mathsf{P}_{\mathsf{trans}}$, and an example $\mathsf{i} \mapsto \mathsf{o}$, the provenance information takes the form of $\mathsf{SP}_0 \leftarrow \mathsf{si}_0, \ldots, \mathsf{SP}_n \leftarrow \mathsf{si}_n$, where (a) each si_j is a subtree of i, and (b) each SP_j is a sub-program of $\mathsf{P}_{\mathsf{trans}}$ that is a `select`, and SP_j produces the output si_j during the execution of $\mathsf{P}_{\mathsf{trans}}(\mathsf{si})$. We call the subtrees si_j the *selected nodes* of the input i. Note that each SP_j may have multiple subtrees si_j and si'_j with $j \neq j'$ such that $\mathsf{SP}_j \leftarrow \mathsf{si}_j$ and $\mathsf{SP}_j \leftarrow \mathsf{si}'_j$. One such case is due to the `MapChildren` operator in Figure 2.2. The lambda function (produced by `transformer`) may have `select` programs that operate over all children of a given node.

Example 5.4.1 *Let us revisit the edits in Example 5.3.1. The transformer* $\mathsf{P}_{\mathsf{trans}}$ *can be represented using* ReFazer*'s DSL as follows:*

```
Tree(CallExpression, [], Cons(
  Tree(DotExpression, [], Cons(
    Tree(Identifier, [TextValue=DependencyResolveUtility], EmptyChildren),
    Cons(select1, EmptyChildren))),
  Cons(select2, select3)))
```

where, `select1`*,* `select2`*, and* `select3` *extract the fragments* X_2*,* X_1*, and* $\mathsf{X}_3, \mathsf{X}_4, \mathsf{X}_5$*,* X_6, X_7 *respectively. Consider this* $\mathsf{P}_{\mathsf{trans}}$ *with this abbreviated example:*

`repository.ResolveDependency(dependency1, args...)` $\mapsto$
`DependencyResolverUtility.ResolveDependency(repository, dependency1, args...)`

The provenance information is given by $\pi =$ { `select1` $\leftarrow$ `ResolveDependency`, `select2` $\leftarrow$ `repository`, `select3` $\leftarrow$ `args`...}.

Anti-Unification. The next step in the algorithm is to compute an anti-unification of inputs and additional positive inputs. Given two inputs i_1 and i_2, the *anti-unification* $\mathsf{i}_1 \bowtie \mathsf{i}_2$ is given by a pair $(\tau, \langle \sigma_1, \sigma_2 \rangle)$ where:

- *template* τ, is an AST with labelled holes $\{\mathsf{h}_0, \ldots, \mathsf{h}_n\}$, and
- two substitutions $\sigma_1, \sigma_2 : \{\mathsf{h}_0, \ldots, \mathsf{h}_n\} \rightarrow \mathbb{T}$ such that $\sigma_1(\tau) = \mathsf{i}_1 \wedge \sigma_2(\tau) = \mathsf{i}_2$.

This definition can be generalized to more than two inputs. For arbitrary number of inputs, we use the notation $\bowtie\{\mathsf{i}_1, \ldots, \mathsf{i}_n\}$. As is standard, we write anti-unification to mean the anti-unification that produces the most specific generalization.

Example 5.4.2 *Consider inputs* $\mathsf{i}_1 = \texttt{if}(\texttt{score} < \texttt{limit})$ *and* $\mathsf{i}_2 = \texttt{if}(\texttt{GetScore}(\texttt{run}) < \texttt{limit})$. *Then the anti-unification* $\bowtie\{\mathsf{i}_1, \mathsf{i}_2\} = \texttt{if}(\mathsf{h}_0 < \texttt{limit}), \langle\{\mathsf{h}_0 \mapsto \texttt{score}\}, \{\mathsf{h}_0 \mapsto \texttt{GetScore}(\texttt{run})\}\rangle$. *It is more specific than any other generalization of* i_1 *and* i_2*, e.g., an anti-unification with template* $\texttt{if}(\mathsf{h}_0 < \mathsf{h}_1)$.

We do not go into the details of the procedure for computing anti-unification but explain the procedure briefly. The procedure is a variant of anti-unification modulo associativity-unity (AU). First, we categorize all possible AST nodes into two different categories, based on the label:

- Fixed arity nodes: These are nodes that always have a fixed number of children. For example, Identifier always has 0 children, CallExpression always has 2 children (function and argument list), and PlusExpression always has 2 children.
- Variable arity nodes: These nodes can have different number of children. For example, ParameterList, Block, and ClassDeclaration. One key observation is that in the AST domain, the children of every variable arity node can be treated as a homogeneous list. That is, no position in the list has a special meaning: every child in a parameter list is a parameter. In contrast, the two children of CallExpression are functionally different.

Now, $\mathsf{i}_1 \bowtie \mathsf{i}_2$ is computed as follows:

- If the roots of i_1 and i_2 have different labels or attributes: $\mathsf{i}_1 \bowtie \mathsf{i}_2 = (\mathsf{h}, (\{\mathsf{h} \mapsto \mathsf{i}_1\}, \{\mathsf{h} \mapsto \mathsf{i}_2\}))$.
- If the root nodes of i_1 and i_2 have the same label label and attributes attrs, and if the nodes are fixed-arity: then $\mathsf{i}_1 \bowtie \mathsf{i}_2 = \mathsf{Tree}(\mathsf{label}, \mathsf{attrs}, \tau_1 \ldots \tau_n), \langle \bigcup_i \sigma_1^i, \bigcup_i \sigma_2^i \rangle$ where (a) $\mathsf{Children}(\mathsf{i}_1) = \mathsf{i}_1^1, \ldots, \mathsf{i}_1^n$ and $\mathsf{Children}(\mathsf{i}_2) = \mathsf{i}_2^1, \ldots, \mathsf{i}_2^n$, and (b) for all $1 \leq j \leq n . \mathsf{i}_1^j \bowtie \mathsf{i}_2^j = (\tau_j, (\sigma_1^j, \sigma_2^j))$

- If the root nodes of i_1 and i_2 have the same label label and are variable arity nodes: Let the children of i_1 and i_2 be $\mathsf{i}_1^1, \ldots, \mathsf{i}_1^n$ and $\mathsf{i}_2^1, \ldots, \mathsf{i}_2^m$, respectively. Then, we compute two lists of node sequences $\mathsf{s}^0, \mathsf{d}_i^0, \mathsf{s}^1 \ldots \mathsf{d}_i^k, \mathsf{s}^k$ for $i \in \{1, 2\}$ such that: (a) The concatenation $\mathsf{s}^0\mathsf{d}_i^0\mathsf{s}^1 \ldots \mathsf{d}_i^k\mathsf{s}^k$ is equal to $\mathsf{i}_1^1, \ldots, \mathsf{i}_1^n$ and $\mathsf{i}_2^1, \ldots, \mathsf{i}_2^n$ for $i = 1$ and $i = 2$, respectively. Note that s^i and d_i^b are nodes that are shared and are different in the two lists, respectively. (b) the combined length of s_i^j is maximized.

Note that some d_i^j may be the empty list nil which acts as the identity for the concatenation operation. Now, the anti-unification $\mathsf{i}_1 \bowtie \mathsf{i}_2 = (\mathsf{Tree}(\mathsf{label}, \mathsf{attrs}, \mathsf{s}^1\mathsf{h}_1 \ldots \mathsf{s}^k), \langle \{\mathsf{h}_i \mapsto \mathsf{d}_1^i \mid 0 \leq i \leq k\}, \{\mathsf{h}_i \mapsto \mathsf{d}_2^i \mid 0 \leq i \leq k\}\rangle)$.

Remark 5.4.1 *The anti-unification of two ASTs* i_1 *and* i_2 *is not uniquely defined. For example, let both* i_1 *and* i_2 *be argument lists with* $\mathsf{i}_1 = (\mathtt{x}, \mathtt{x})$ *and* $\mathsf{i}_2 = (\mathtt{x})$ *where* $\mathtt{x}$ *is a variable. Now,* $\mathsf{i}_1 \bowtie \mathsf{i}_2$ *is computed as per the third case above. As per the definition, we have two options for the result: (a)* $((\mathtt{x}, \mathsf{h}), \langle \{\mathsf{h} \mapsto \mathtt{x}\}, \{\mathsf{h} \mapsto \mathsf{nil}\}\rangle$*, or (b)* $((\mathsf{h}, \mathtt{x}), \langle \{\mathsf{h} \mapsto \mathtt{x}\}, \{\mathsf{h} \mapsto \mathsf{nil}\}\rangle$*. That is, it is unclear if the* $\mathtt{x}$ *in* i_2 *matches with the first or the second* $\mathtt{x}$ *in* i_1*. This issue can be resolved by using more advanced anti-unification techniques.*

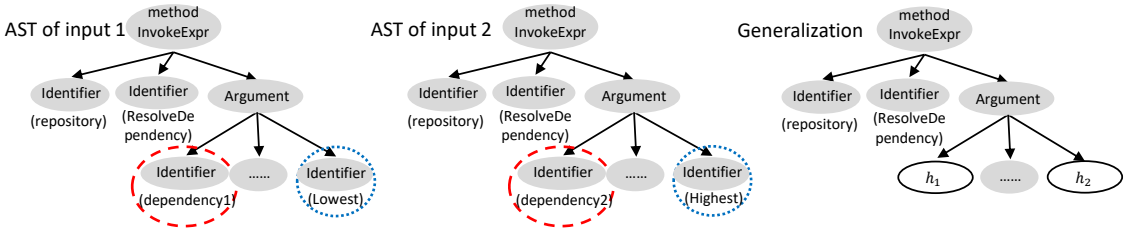


Figure 5.4: The partial AST of two inputs shown in Figure 5.1a and 5.1c, and their generalization.

For our use case, we do not consider the general notion of anti-unification, but *anti-unification modulo provenance*. Consider inputs i_1 and i_2, and provenance information π derived from evaluation a transformation $\mathsf{P}_{\mathsf{trans}}$ on i. The anti-unification modulo provenance $\mathsf{i}_1 \bowtie_\pi \mathsf{i}_2$ is given by $(\tau, \langle \sigma_1, \sigma_2 \rangle)$ where:

- $(\tau, \langle \sigma_1, \sigma_2 \rangle)$ is an anti-unification of i_1 and i_2, i.e., $\sigma_1(\tau) = \mathsf{i}_1$ and $\sigma_2(\tau) = \mathsf{i}_2$;

- For each substitution $(\mathsf{h} \mapsto \mathsf{si}) \in \sigma_1$, either (a) si is a selected node, i.e., $(\mathsf{SP} \leftarrow \mathsf{si}) \in \pi$ for some SP; or (b) si has no ancestors or descendants that are selected nodes. Note that this condition is only relevant for σ_1 as the provenance π is derived from evaluating a transformation on i.

The additional constraint on the substitutions makes anti-unification modulo provenance be undefined in certain cases (see Example 5.4.3).

Example 5.4.3 *Consider the input* $\mathsf{i}_1 = \texttt{score} < \texttt{limit}$ *from the example* $\texttt{score} < \texttt{limit} \mapsto \texttt{IsValid}(\texttt{score})$ *and the additional input* $\mathsf{i}_2 = \texttt{GetScore}(\texttt{run}) < \texttt{limit}$. *Given* i_1 *and* i_2*, the anti-unification procedure generates substitutions* $\sigma_1 = \{\mathsf{h} \mapsto \texttt{score}\}$ *and* $\sigma_2 = \{\mathsf{h} \mapsto \texttt{GetScore}(\texttt{run})\}$ *with the template* $\mathsf{h} < \texttt{limit}$. *Given the input-output example and its corresponding transformation, the provenance procedure produces* $\pi = \{\mathsf{SP} \leftarrow \texttt{score}\}$ *for some sub-program* SP *that is a* `select` *operation. Note that* `score` *in* $\sigma_1 = \{\mathsf{h} \mapsto \texttt{score}\}$ *is a selected node in* π*, and thus, the anti-unification modulo* π *of* i_1 *and* i_2 *exists. Now, consider another input* $\mathsf{i}_3 = \texttt{score} == \texttt{GetScore}(\texttt{run})$. *Given* i_1 *and* i_3*, the anti-unification procedure generates substitutions* $\sigma_1' = \{\mathsf{h} \mapsto \texttt{score} < \texttt{limit}\}$ *and* $\sigma_3' = \{\mathsf{h} \mapsto \texttt{score} == \texttt{GetScore}(\texttt{run})\}$ *with the template* h. *Here, the root node of* i_1 *is* `LessThanExpression` *and of* i_3 *is* `EqualsExpression`*: hence, the expressions cannot be unified further. In this case, the condition for the anti-unification modulo* π *does not hold, as the substitution* $\mathsf{h} \mapsto \mathsf{i}_1$ *returns the root node of* i_1 *which is not a selected node, but has a descendant that is a selected node. Thus, the anti-unification modulo* π *of* i_1 *and* i_3 *does not exist.*

Intuitively, we are trying to match "important parts" (here, selected nodes) of i_1 *with equivalent parts in* i_2 *and* i_3. *We can match the nodes* `score` *in* i_1 *and* $\texttt{GetScore}(\texttt{run})$ *in* i_2 *as they are represented by the same hole in the anti-unification, and thus, they are compatible. Conversely, we cannot match* `score` *in* i_1 *and* `score` *in* i_3*, because, even though they are equal, there is no hole in the anti-unification of* i_1 *and* i_3 *that maps to them. Thus, they are incompatible.*

Completing the Procedure. Given the above anti-unification modulo provenance computation, we produce the potential outputs for all additional positive inputs PI. For producing these outputs, we use an evaluation process that uses

an input i from an example and an additional input pi. This process is denoted as $\mathsf{Evaluate}^*(\mathsf{P}_{\mathsf{trans}}, \mathsf{pi}, \mathsf{i})$. Let σ and σ' be the substitutions for i and pi in the anti-unification modulo provenance, respectively. We evaluate $\mathsf{P}_{\mathsf{trans}}$ on pi as follows:

- For every sub-program SP of $\mathsf{P}_{\mathsf{trans}}$ which is a `select`, let $\mathsf{SP} \leftarrow \mathsf{si} \in \pi$. Then, the evaluation value is set to $\sigma'(\sigma^{-1}(\mathsf{si}))$.
- For every sub-program SP of $\mathsf{P}_{\mathsf{trans}}$ which is not a `select`, we evaluate the value by applying the top level operator on the evaluated values of the children, as usual.

Example 5.4.4 *Consider the first input in Figure 5.1a and 5.1c, anti-unification generates* $\sigma_1 = \{\mathsf{h}_1 \mapsto \texttt{dependency}, \mathsf{h}_2 \mapsto \texttt{Lowest}\}$ *and* $\sigma_2 = \{\mathsf{h}_1 \mapsto \texttt{dependency2}, \mathsf{h}_2 \mapsto \texttt{Highest}\}$. *In order to produce an output for the additional positive input in 5.1c, we apply* $\sigma_2(\sigma_1^{-1}(\mathsf{si}))$ *to every* $\mathsf{SP} \leftarrow \mathsf{si} \in \pi$. *The elements of interest in* π *are:* `select1` $\leftarrow$ `dependency` *and* `select2` $\leftarrow$ `Lowest` *for some select sub-programs* `select1` *and* `select2`. *Now, we have* $\sigma_2(\sigma_1^{-1}(\texttt{dependency})) = \texttt{dependency1}$ *and* $\sigma_2(\sigma_1^{-1}(\texttt{Lowest})) = \texttt{Highest}$. *Using these values as the evaluation results of* `select1` *and* `select2` *and continuing evaluation, we end up with the output:*

`DependancyResolverUtility.ResolveDependency(dependency1, ..., Highest)`.

Once we have the outputs for the additional positive inputs, we provide the given examples along with the new examples generated from additional positive inputs to the transformer synthesis component of ReFazer.

Theorem 5.4.2 (Soundness) *Algorithm 5 is sound: if a program* P *is returned, then (a)* $\forall \mathsf{i} \mapsto \mathsf{o} \in \mathsf{Examples.P}(\mathsf{i}) = \mathsf{o}$, *(b)* $\forall \mathsf{pi} \in \mathsf{PI.P}(\mathsf{pi}) \neq \bot$, *and (c)* $\forall \mathsf{ni} \in \mathsf{NI.P}(\mathsf{ni}) = \bot$.

The proof follows from the use of $\mathsf{ReFazer}_{\mathsf{trans}}$ and $\mathsf{ReFazer}_{\mathsf{guard}}$ in lines 17 and 2, respectively. Note that, it is possible that the inferred output po for the additional positive input pi is incorrect. In this situation, the user can add a new input-output example (positive or negative) that has the same input that was incorrectly classified. We will ignore the additional input i if there exists an input from the input-output examples that is same as i.

Remark 5.4.3 (Completeness of Algorithm 5) *Algorithm 5 is not complete, i.e., it may not return a program even when one satisfying all requirements exists. This is an intentional choice. Consider* $\mathsf{Examples} = \{\text{“}(\texttt{temp} - 32) * (5/9)\text{”} \mapsto \text{“}\texttt{FtoC(temp)}\text{”}\}$, $\mathsf{PI} = \{\text{“}\texttt{x} = \texttt{x} + 1;\text{”}\}$, *and* $\mathsf{NI} = \emptyset$, *the input of the example and the additional positive input are not logically related. However, there exists a program that is correct, i.e., the program that returns the constant tree* “`FtoC(temp)`”. *In any practical scenario, this constant program is very unlikely to be the intended program. Hence, we explicitly make the choice of incompleteness.*

5.4.2 Feedback-Driven Semi-Supervised Synthesis

Algorithm 6: Feedback-driven semi-supervised synthesis

Input: Feedback oracle
$\mathsf{Feedback} : \mathcal{P} \times (\mathbb{T} \not\to \mathbb{T}) \times \mathbf{2}^{\mathbb{T}} \times \mathbf{2}^{\mathbb{T}} \times \mathbf{2}^{\mathbb{T}} \to (\mathbb{T} \to [-\infty, \infty])$.
Input: Semi-supervised synthesis engine $\mathsf{SynthesisEngine}$.
Input: Pool of available inputs $\mathsf{InputPool}$.
Input: Initial examples $\mathsf{Examples}$, positive inputs PI, and negative inputs NI.
Input: Thresholds $p, n \in \mathbb{R}$.

while `true` **do**
  $\mathsf{P} \leftarrow \mathsf{SynthesisEngine}(\mathsf{Examples}, \mathsf{PI}, \mathsf{NI})$;
  Notify user of current suggestions: $\{\mathsf{i} \mapsto \mathsf{o} \mid \mathsf{i} \in \mathsf{InputPool} \wedge \mathsf{o} = \mathsf{P}(\mathsf{i}) \wedge \mathsf{o} \neq \bot\}$;
  $\mathsf{Rew} \leftarrow \mathsf{Feedback}(\mathsf{P}, \mathsf{Examples}, \mathsf{InputPool}, \mathsf{PI}, \mathsf{NI})$;
  $\mathsf{PI}' \leftarrow \{\mathsf{i} \in \mathsf{InputPool} \mid \mathsf{Rew}(\mathsf{i}) > p\}$;
  $\mathsf{NI}' \leftarrow \{\mathsf{i} \in \mathsf{InputPool} \mid \mathsf{Rew}(\mathsf{i}) < n\}$;
  if $*$ **then**
    $\mathsf{PI} \leftarrow \mathsf{PI} \cup \mathsf{pi}'$ where pi' is an arbitrary input from PI';
  else
    $\mathsf{NI} \leftarrow \mathsf{NI} \cup \mathsf{ni}'$ where ni' is an arbitrary input from NI';
  end
end

Algorithm 12 presents a procedure for the feedback-driven semi-supervised synthesis problem that closely follows Figure 5.3. It takes the following as input: (a) A feedback oracle $\mathsf{Feedback}$ that represents the user and the environment. The feedback oracle takes as input a program P, a set of examples $\mathsf{Examples}$, an input pool $\mathsf{InputPool}$, positive inputs PI, and negative inputs NI, and produces a reward function $\mathsf{Rew} : \mathsf{InputPool} \to [-\infty, \infty]$. Informally, the feedback oracle checks the whole state of the process, and produces rewards for inputs from the pool. (b) A

semi-supervised synthesis procedure SynthesisEngine depicted in Algorithm 5. (c) An input pool, an initial non-empty set of examples, a set of positive inputs, and a set of negative inputs.

In addition, the algorithm uses the thresholds p and n to determine if an input from the input pool should be added to either the positive or negative inputs. These thresholds are dependant on the application scenario and the Feedback oracle. In Section 5.5, we present three different application scenarios and the choice of p and n for them. For the Feedback oracle, we present two different oracles $\mathsf{Feedback}_{\mathsf{user}}$ and $\mathsf{Feedback}_{\mathsf{auto}}$. In the application scenarios, these oracles are combined in different ways to obtain application specific feedback oracles.

User-Driven Feedback Oracle. The *user-driven feedback oracle* $\mathsf{Feedback}_{\mathsf{user}}$ represents the user of the application. In different interfaces, the feedback from the user can take different forms, each of which can be converted to a reward function $\mathsf{Rew}_{\mathsf{U}} : \mathsf{InputPool} \to [-\infty, +\infty]$. We have the following two cases (Section 5.5):

- The user explicitly provides new positive inputs PI' and negative inputs NI'. We convert this feedback into the reward function $\mathsf{Rew}_{\mathsf{U}}$ by setting $\forall \mathsf{pi} \in \mathsf{PI}'.\mathsf{Rew}_{\mathsf{U}}(\mathsf{pi}) = +\infty$, $\forall \mathsf{ni} \in \mathsf{NI}'.\mathsf{Rew}_{\mathsf{U}}(\mathsf{ni}) = -\infty$, and $\mathsf{Rew}_{\mathsf{U}}(\mathsf{i}) = 0$ for all other inputs in InputPool.

- The user provides a set of candidate positive inputs PI^* with the intent that the transformation should apply to one of these candidate positive inputs. For example, a set of candidate positive inputs could be a set of ASTs that contain the cursor location in a file. We give a constant reward to all the nodes in PI^*, i.e., we have $\forall \mathsf{pi} \in \mathsf{PI}^*.\mathsf{Rew}_{\mathsf{U}}(\mathsf{pi}) = C$ where $0 < C < +\infty$. In our implementation, we set C as 2.

With richer user interfaces, we could consider more complex forms of $\mathsf{Feedback}_{\mathsf{user}}$ oracle.

Fully Automated Feedback Oracle. *Fully automated feedback oracle* $\mathsf{Feedback}_{\mathsf{auto}}$ represents the environment the synthesizer is operating in. It can include a number of independent components only restricted by the available tools in the environment the synthesizer is running in. For example, if a synthesizer is running inside an IDE, the oracle could use the compiler or the version control history. Algorithm 7

presents a basic oracle that reuses the provenance and anti-unification computation from the semi-supervised synthesis engine, and, uses the scoring function Score on guards and a bound $\mathsf{threshold}_g$ on scores. The scoring function and bound we use are the same as in BluePencil [87], which in turn takes the scoring function from [109]. In practice, the feedback loop in Algorithm 12 can be optimized by sharing the provenance computation and anti-unification across the synthesis engine and the $\mathsf{Feedback}_{\mathsf{auto}}$ oracle.

Algorithm 7: The fully automated feedback oracle $\mathsf{Feedback}_{\mathsf{user}}$

Input: Compiler $\mathsf{Compiler} : t \to \mathbb{B}$ or $\bot$ if compiler is not available
Input: Distance metric $\mathsf{Distance} : \mathbb{T} \times \mathbb{T} \to \mathbb{R}^{\geq 0}$
Input: Program $\mathsf{P} = (\mathsf{P}_{\mathsf{guard}}, \mathsf{P}_{\mathsf{trans}})$
Input: Examples $\mathsf{Examples} : \mathbb{T} \times \mathbb{T}$
Input: Input pool $\mathsf{InputPool}$
Input: Positive Examples PI, Negative Examples NI
Output: Rewards function $\mathsf{Rew}_{\mathsf{E}} : \mathsf{InputPool} \not\to [-\infty, +\infty]$

```
1  i* ↦ o* ← arbitrary example in Examples;
2  π ← Provenance(i* ↦ o*, P_trans);
3  Rew_E ← ∅;
4  for i ∈ InputPool do
5      if P(i) ≠ ⊥ ∧ Compiler ≠ ⊥ ∧ Compiler(P(i)) = false then
6          Rew_E ← Rew_E ∪ {i ↦ −∞};
7          continue;
8      end
9      guard ← ReFazer_guard({i | i ↦ o ∈ Examples} ∪ PI ∪ {i}, NI);
10     if Score(guard) < threshold_g then
11         Rew_E ← Rew_E ∪ {i ↦ −∞};
12         continue;
13     end
14     d ← 1 − Distance(i, i*);
15     Rew_E ← Rew_E ∪ {i ↦ d};
16 end
17 return Rew_E
```

Algorithm 7 works as follows. For each candidate (positive or negative) additional input i in the input pool:

- If the program P on i produces an output and that output cannot be compiled, reward is $-\infty$ (line 5). Though compilation can be expensive, in practice, IDEs allow for efficient incremental compilation. Further, this step is not as

expensive as P typically does not produce an output, i.e. $\mathsf{P}_{\mathsf{guard}}(\mathsf{i}) = \mathsf{false}$, for most $\mathsf{i} \in \mathsf{InputPool}$.

- Otherwise, we synthesize a guard that matches the examples and positive inputs along with the candidate input i using $\mathsf{ReFazer}_{\mathsf{guard}}$. Similar to BluePencil [87], we bound the score of the guard with a threshold to avoid overly general guards, which are almost never the intended one (line 9).

- Otherwise, we compute the distance between the candidate input i and an example input i^*, using a $\mathsf{Distance}$ function, i.e. $\mathsf{Rew}_{\mathsf{E}} = 1 - \mathsf{Distance}(\mathsf{i}, \mathsf{i}^*)$, where $\mathsf{Distance}(\mathsf{i}, \mathsf{i}^*) \in [0, 1]$ (line 12). The $\mathsf{Distance}$ function is explained in detail below.

The Distance Function. Consider an input i^* that comes from an example $\mathsf{i}^* \mapsto \mathsf{o}^*$, and a candidate additional input i. Intuitively, we want to give a high reward if i is similar to i^*. However, we need a more involved notion of similarity than standard clone detection techniques.

Example 5.4.5 *Consider the example* `if(score < limit)` $\mapsto$ `if(IsValid(score))` *and the candidate additional input* `if(GetScore(run) < limit)`*. A tree-based clone-detection technique would not classify the above two inputs as clones given the high difference between* `score` *and* `GetScore(run)`*. However, as we described in Example 5.4.3, the anti-unification modulo π of these inputs tells us that (i)* `score` *is a relevant part of the input since it also appears in the output, and (ii)* `score` *and* `GetScore(run)` *are compatible since there is a hole in the anti-unification that maps to these nodes.*

Given that we already have this information about the compatibility of these subtrees, we "relax" the tree distance comparison between these two inputs. Rather than comparing the concrete subtrees, we abstract them using a technique called d-caps [91, 24]. For a $d \geq 0$, the d-cap of a node replaces all the sub-nodes at depth d with holes. For instance, when $d = 1$, instead of comparing `score` *and* `GetScore(run)`*, we compare the nodes (with no children)* Identifier *and* `CallExpression`*, which are their corresponding root nodes. Note that expression* `score` *is shorthand for a node with label* Identifier*, attributes* $\{\mathsf{TextValue} \mapsto \mathsf{score}\}$*, and no children. Both the subtrees*

have been truncated to a depth of 1. *This "loosens" the comparison between these nodes, and returns a smaller difference value.*

Further, consider candidate additional input `if(score > UnrelatedFunction())`. *The difference between the two inputs is* `< limit` *and* `> SomeUnrelatedFunction()`. *These two fragments are not directly used in the output, and thus we cannot rely on the anti-unification modulo* π *to assess their compatibility. Hence, it is essential that we include this particular difference in the computation of distance.*

Concretely, our Distance function represents the d-cap replaced input as numerical vectors and uses the Euclidean distance between these vectors to represent the distance between the trees, similar to Deckard [50], a clone detection technique. The distance between the two inputs i_1 and i_2 can then be formally defined as follows:

$$\begin{aligned}\mathsf{Distance}(\mathsf{i}_1, \mathsf{i}_2) = \mathsf{CloneDetection}(\sigma_1^\dagger(\tau), \sigma_2^\dagger(\tau)) \text{ where} \\ (\tau, \langle\sigma_1, \sigma_2\rangle) = \mathsf{i}_1 \bowtie_\pi \mathsf{i}_2 \\ \sigma_1^\dagger, \sigma_2^\dagger = \mathsf{DCapModuloProvenance}(\sigma_1, \sigma_2, \pi)\end{aligned}$$

Here, DCapModuloProvenance replaces each substitution for a selected subtree with its *d-cap*. Formally, $\sigma_i^\dagger(\mathsf{h})$ is equal to: (a) the d-cap of $\sigma_i(\mathsf{h})$ if $\sigma_i(\mathsf{h})$ is a selected node, and (b) $\sigma_i(\mathsf{h})$ otherwise.

5.5 Applications of Semi-Supervised Synthesis

In this section, we present three practical applications of semi-supervised synthesis in the domain of C# program transformations. They allow different types of feedback to produce additional positive inputs to the semi-supervised synthesizer. To implement the semi-supervised synthesis algorithm (Algorithm 5), we leverage the Transformation.Tree API available in the PROSE SDK as a concrete implementation of ReFazer. Additionally, in all applications, we use all the AST nodes available in the source code file as inputs for the input pool. In our implementation, we use untyped ASTs, i.e., each node in the AST does not have the type of the corresponding expression as an attribute. While our techniques are able to handle typed ASTs, performing type inference on every edit can incur performance penalties.

5.5.1 ReFazer* User-Provided Feedback about Additional Inputs

ReFazer* uses the user-driven feedback oracle to identify positive inputs to the semi-supervised synthesizer. The target for ReFazer* is applications where a developer is providing examples manually. To illustrate this application, consider our motivating example shown in Figure 5.1. For the first false negative (Figure 5.1c), instead of manually performing the edit to give another example, the developer can provide feedback to the system by indicating that the location (text selection representing the input AST) should have been modified. ReFazer* uses the feedback to create a positive input and generalize the transformation. After that, ReFazer* produces suggestions to two out of the three false negatives. The developer can follow the same process to fix the other false negative. In terms of the feedback oracles from the previous section, $\mathsf{Feedback}_{\mathsf{user}}$ returns a reward function $\mathsf{Rew}_{\mathsf{U}}$ that is $+\infty$ on the additional positive input the developer has provided, and 0 everywhere else. Further, we pick the thresholds p and n to both have the value 0. Similarly, if ReFazer* produces a false positive on some location, developers could provide feedback to the system by indicating that this location (text selection and press predefined shortcut) should not be modified. With this feedback, $\mathsf{Feedback}_{\mathsf{user}}$ returns a reward $-\infty$ on the additional negative input provided by developers. Correspondingly, ReFazer* will refine the synthesized transformation with additional examples to avoid generating similar false positives.

ReFazer* requires the developer to enter a special mode to provide examples and feedback to the system. While this interaction gives more control to the developer, it may also prevent developers from using it due to discoverability problems [87]. Next, we describe two other *modeless* applications of our technique that do not require explicitly providing examples and feedback.

5.5.2 BluePencil$_{\mathsf{cur}}$ Semi-Automated Feedback Based on Cursor Position

For our second application, we instantiated the BluePencil algorithm [87] using our semi-supervised synthesizer as the PBE synthesizer. BluePencil works in the background of an editor. While the developer edits the code, the system

infers examples of repetitive edits from the history of edits, and it uses a synthesizer to learn program transformations for these edits. The original algorithm does not consider sets of input-output examples of size one, as they do not indicate repetitive changes. We modified this constraint to allow the system to use BluePencil$_{\mathsf{cur}}$ to learn transformations from just one example and one additional positive input.

To enable the completely modeless interaction, BluePencil$_{\mathsf{cur}}$ uses both user-driven and fully automated oracles to produce feedback. The former leverages the cursor position to collect implicit feedback from the developer. Note that the developer is not actively providing feedback—it is completely transparent to the developer, and is inferred automatically. Intuitively, the cursor suggests that the developer is interested in that part of the code and may want to edit it.

However, the cursor location is very ambiguous: the subtree the developer is likely to edit can be any subtree that contains the cursor location. Consider the false negative shown in Figure 5.1c. Suppose the developer places the cursor location at the beginning of the line. There are many subtrees that include this location, including the ones corresponding to the following code fragments: `repository` and `repository.ResolveDependency(...)`. The latter is the input that should be classified as a positive input. The $\mathsf{Feedback}_{\mathsf{user}}$ oracle returns a reward function that gives a positive score ($\mathsf{Rew}_{\mathsf{U}}$) to all subtrees that include the position defined by the cursor. We also use feedback from the $\mathsf{Feedback}_{\mathsf{auto}}$ oracle described in Section 5.4.2 to further disambiguate the cursor location. Intuitively, $\mathsf{Feedback}_{\mathsf{auto}}$ will provide positive rewards ($\mathsf{Rew}_{\mathsf{E}}$) to the nodes that are "similar" to the example inputs. Finally, we regard inputs with $\mathsf{Rew}_{\mathsf{U}}(i) * \mathsf{Rew}_{\mathsf{E}}(i) > p$ as positive inputs and inputs with $\mathsf{Rew}_{\mathsf{U}}(i) * \mathsf{Rew}_{\mathsf{E}}(i) < n$ as negative inputs.

We implement BluePencil$_{\mathsf{cur}}$ as a Visual Studio extension. Figure 5.2 shows the extension in action. As soon as the developer places the cursor in the location related to the false negative, BluePencil$_{\mathsf{cur}}$ uses the semi-supervised feedback synthesis to improve the transformation. The new transformation produces an *auto-completion* suggestion for the current location (see Figure 5.2). In this setting, we are using the user-driven feedback and the automated feedback to more precisely pick the additional positive input. However, there are many settings where it is infeasible to obtain any feedback from the user. We discuss this case in the next section.

5.5.3 BluePencil$_{\text{auto}}$ Fully Automated Feedback Based on All Inputs in the Source Code

Our last application (BluePencil$_{\text{auto}}$) uses fully automated feedback to identify positive inputs without any explicit or implicit feedback from developers. The motivation for this application is that the developers may not be aware of all locations that must be changed or they may want to apply the edits in bulk. We also implemented BluePencil$_{\text{auto}}$ on top of BluePencil. We restricted this application to synthesis tasks that have at least two input-output examples.

Consider again our motivating example (Figure 5.1). As soon as the developer finishes the first two edits (Figure 5.1a), BluePencil$_{\text{auto}}$ automatically identifies the inputs in Figure 5.1c as positive inputs and synthesizes the correct transformation. Now, if the developer is unaware of the other locations, the tool still produces suggestions at these places. These suggestions may then be used to automatically prompt the developer to make these additional edits. Another scenario is as follows: after the two edits, the developer creates a *pull request*. The tool can now be run as an automated reviewer (see, for example, [7]) to suggest changes to the pull request.

5.6 Evaluation

In this section, we present our evaluation of the proposed approach in terms of effectiveness and efficiency. In particular, we evaluate our technique with respect to the following research questions:

RQ1 What is the effectiveness of ReFazer* in generating correct code transformations? We hypothesize that user-provided positive inputs should help our synthesis engine learn better transformations. We evaluate the quality of the synthesized transformation with and without additional positive inputs by measuring the number of false positives (incorrect suggestions) and false negatives (missing suggestions) produced.

RQ2 What is the effectiveness of the reward calculation function? The reward calculation function needs to precisely identify valid additional inputs to avoid generating many false positives or false negatives. We evaluate our

reward calculation function by comparing it with two baseline approaches: *no validation* and *clone detection*.

RQ3 Given a cursor location, what is the effectiveness and efficiency of BluePencil$_{\mathsf{cur}}$?

BluePencil$_{\mathsf{cur}}$ should generate edit suggestions at the cursor location efficiently enough to be usable as an auto-completion feature in an IDE, while still maintaining the quality of suggestions. Given cursor locations, we measure the number of false positives and negatives produced by BluePencil$_{\mathsf{cur}}$, and the time taken to produce the suggestions.

RQ4 How do BluePencil$_{\mathsf{cur}}$ and BluePencil$_{\mathsf{auto}}$ compare to BluePencil?

BluePencil$_{\mathsf{cur}}$ and BluePencil$_{\mathsf{auto}}$ are both built on top of BluePencil, and they aim at reducing the number of examples developers need to provide. By simulating a developer performing repetitive edits using these tools, we compare how much information (examples and locations) is required by each one of them.

5.6.1 Benchmark Suite

We collected 86 occurrences of real life code editing sessions containing repetitive edits. These scenarios were collected from developers at Microsoft spanning multiple teams during the internal testing phase of the Microsoft Visual Studio IntelliCode suggestions feature (BluePencil).

Each session consists of a list of program versions representing the history of the program content as the user makes edits. For each session, we manually generated the ground truth data containing the *number of repetitive edits*, the *version ids* before and after each repetitive edit, and the *locations* and *content change* for each repetitive edit. Each editing session contains one or multiple sequences of repetitive edit transformations, with each sequence containing at least two repetitive edits. Each session also contained noise, i.e., edits that are not a part of any repetitive sequence. Figure 5.5 shows the number of repetitive edits in different program editing sessions, where the x-axis presents the number of repetitive edits and y-axis gives the number of editing sessions. For instance, there are 25 (around 30%) editing sessions

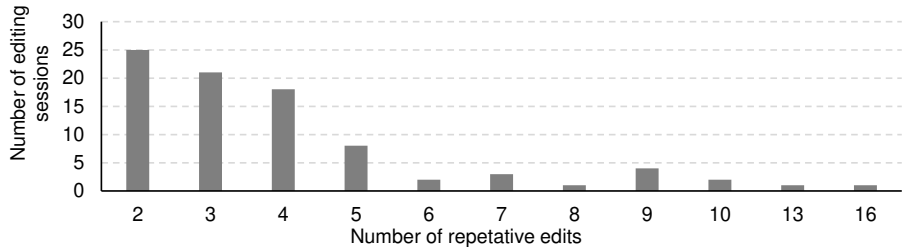


Figure 5.5: The distribution of number of repetitive edits across the programs

with 2 repetitive edits. This high percentage also motivates the need for a technique that automates edits with fewer examples, ideally 1 example. Techniques such as BluePencil that require at least two examples cannot generate any suggestions for cases with just 2 repetitive edits in the session. The average number of repetitive edits is 4.07 while the largest number is 16. The benchmark suite contains a variety of edits, from small edits that change only a single program statement to large edits that modify code blocks.

All the experiments were conducted on a machine equipped with Inter Core i7-8700T CPU @ 2.4GHz, 32GB memory running 64-bit Windows 10 Enterprise.

5.6.2 Effectiveness of Semi-Supervised Synthesis

In the scenario where a developer manually indicates an additional positive input for a repetitive transformation, we evaluate the effectiveness of ReFazer* by measuring its precision and recall in generating correct suggestions. In this evaluation, we use ReFazer [109] as our baseline.

Experimental Setup. In each program editing session, we first manually extract all the repetitive edits. For a session with M repetitive edits, we provide N edits as examples for the synthesis engine, and the remaining repetitive edits in this session are used for testing. We set $N < M$ to ensure there is at least one edit that can be used for testing, further, we limit N up to three. Considering that users could perform the repetitive edits in any order, we consider all combinations when choosing the examples. For instance, for a session with three repetitive edits (e_1, e_2, e_3), the users could manually complete e_1 and ReFazer* automates e_2 and e_3. The user could also complete e_3, and ReFazer* automates e_1 and e_2. Different edits contain slightly different information: the result of the synthesizer

Table 5.1: The effectiveness of semi-supervised synthesis.

Examples (N)	Session	Edit	Scenario	ReFazer		ReFazer*	
				Precision	Recall	Precision	Recall
One	86	350	1400	100.00%	26.71%	96.01%	100.00%
Two	61	300	3664	99.65%	77.26%	98.58%	99.94%
Three	40	237	7578	99.88%	89.10%	99.72%	99.99%

depends not only on the number of examples but also on which examples were used. We try all combinations of N examples to avoid any bias introduced by picking a particular order. For an editing session with M repetitive edits, there are $C(M, N)$ combinations when choosing the N examples. For instance, for a program edit session with four repetitive edits, if two edits are provided to the PBE engine as examples, there are $C(4, 2) = 6$ combinations. Given a combination of N examples to the PBE engine, we then create a set of testing scenarios where the N edits are provided to PBE engine as examples, and one of the $M - N$ other edits is used for testing. Therefore, for an editing session with M repetitive edits, we create $C(M, N) * (M - N)$ scenarios. In each test, ReFazer* also takes the input from testing edit as additional positive input. We then compare the output of the synthesized transformation on the test input against the test output. We calculate the precision and recall of ReFazer and ReFazer* by measuring the number of false positives, false negatives, and true positives produced in all the scenarios.

Experimental Parameters. In this experiment, we set $\mathsf{Rew_U}(\mathsf{pi}) = +\infty$ for the user-provided positive input $\mathsf{pi} \in \mathsf{PI}$ and $\mathsf{Rew_U}(\mathsf{ni}) = -\infty$ for the user-provided negative input $\mathsf{ni} \in \mathsf{NI}$. Further, we set both p and n in Algorithm 12 as 0.

Evaluation Results. Table 5.1 presents our evaluation results of traditional ReFazer and ReFazer*. The first column displays the number of examples provided to PBE engine, while the Session column shows the number of program editing sessions. Edit and Scenario columns display the number of edits and scenarios, respectively. The more examples the PBE engine takes, the more scenarios we create because there are more combinations when choosing examples. By comparing the different number of examples, ReFazer produces much better results (recall) with more examples (from 26.71% with one example to 89.10% with three examples). This is because the synthesis engine can learn how to generalize the transformation with

more examples. The precision is always high because ReFazer always learns the most specific transformation which is unlikely to produce false positives. However, too specific transformations easily result in false negatives. Especially, the recall with one example is just 26.71%, which highlights the challenges of synthesizing a high-quality transformation with fewer examples. In contrast, ReFazer* significantly improves the recall regardless of the number of examples, while maintaining the high precision (slightly lower). ReFazer* can generate better results because the additional input helps synthesize a more suitably generalized transformation. Specifically, we achieve 100% recall and >96% precision with only one example, which can release the burden of users from providing multiple repetitive edit examples. Compared to ReFazer, we generate a few more false positives. The nature of these additional false positives is discussed in Section 5.6.6.

> ReFazer* significantly improves the recall of ReFazer while retaining the high precision in generating correct suggestions. Even by taking one example as input, ReFazer* achieves more than 96% precision and 100% recall.

5.6.3 Effectiveness of Reward Calculation Function

Our second experiment evaluates the effectiveness of the proposed reward calculation function. The reward calculation function determines whether a node is an additional positive or negative input for the feedback system. In this section, we evaluate its effectiveness in identifying additional positive inputs by comparing with two baseline approaches: *No validation* and *clone detection*.

- *No validation*: This baseline regards any node as an additional positive input. Hence, we set $\mathsf{Rew}(\mathsf{i}) = +\infty$ for all nodes in the input pool.
- *Clone detection*: Given an edit $\mathsf{i}^* \mapsto o^*$ and one additional node i, we determine whether i is an positive additional input by calculating the normalized distance between i^* and i using clone detection techniques, i.e. $\mathsf{Rew}(\mathsf{i}) = 1 - \mathsf{CloneDetection}(\mathsf{i}, \mathsf{i}^*)$. Here, we use the approach proposed by [50] without the use of the d-cap modulo provenance from Section 5.4.2.

Table 5.2: The effectiveness of the reward calculation function.

Sessions	# pNodes	# nNodes	No validation		Clone detection		Reward function	
			# false positive	# false negative	# false positive	# false negative	# false positive	# false negative
86	265	243417	9055	7	8	111	11	14

- *Reward function based on Distance*: Given edit $\mathsf{i}^* \mapsto o^*$ and additional node i, we use our proposed approach in Algorithm 7 and Section 5.4.2 to calculate the reward score for i.

Experimental Setup. In each program editing session, we select the first edit as the example $\mathsf{i}^* \mapsto o^*$ for the PBE engine. We then create a set of additional inputs to test whether the techniques above can correctly classify each input i in this set as positive or negative. To create this set, we select the inputs of the remaining edits as positive inputs pNodes and all the remaining subtrees from the document that should not be transformed by the synthesized transformation as negative inputs nNodes. We measure the false positives and negatives produced on both pNodes and nNodes by the different approaches.

Experimental Parameters. In this experiment, we set p and n in Algorithm 12 as 0.7 and 0.1, respectively. Specifically, we regard input i as a positive input if $\mathsf{Rew}(\mathsf{i}) > 0.7$ and a negative input if $\mathsf{Rew}(\mathsf{i}) < 0.1$. Further, we set $d = 2$ for d-cap replacement (section 5.4.2).

Evaluation Results. Table 5.2 shows the evaluation results. By regarding any node as an additional positive input, the synthesis engine can successfully generate suggestions for many of them. However, it also generates a large number of false positives (9055), which demonstrates the importance of the additional input validation. If we validate the additional input using existing clone detection (Column Clone detection), the false positive rate is significantly reduced. However, it introduces more false negatives because the clone detection is too strict when comparing two inputs as shown in Section 5.4.2. Considering the fact that we fail to generate suggestions on more than 40% (111 out of 265) of pNodes, the clone detection technique is also not acceptable. The last two columns show the evaluation result of our reward calculation function. We also significantly reduce the number of false positives and we do not introduce too many false negatives. Our reward calculation

Table 5.3: The effectiveness of BluePencil$_{cur}$ when given the history edit trace and the cursor location.

Scenarios	Suggestion	False Positive	False Negative	Precision	Recall	Time (ms)
295	291	1	3	99.66%	98.98%	51.83 (avg)

function results in 3 more false positives than clone detection. The underlying reason will be analyzed in the discussion section.

> The proposed additional input validation can help reduce false positives. Further, it also generates fewer false negatives than existing clone detection techniques.

5.6.4 The Effectiveness and Efficiency of Semi-Automated Feedback

To evaluate the effectiveness and efficiency of BluePencil$_{cur}$, we measure the false positive and false negatives produced at the cursor location by simulating the program editing process of developers.

Experimental Setup. Recall that all the program versions are recorded in form of $\{v_1, v_2, v_3 \ ... \ v_i \ ... \ v_n\}$ on each program editing session. We could easily reproduce the editing steps by going through all the history versions one by one. From the second edit in each editing session (users need to manually complete the first edit), we feed the history versions before edit e_i and the edit location of e_i to BluePencil$_{cur}$. The history versions include at least one repetitive edit (e.g. e_1) and some irrelevant edits (noise). We randomly select a location from the range of edit location to simulate the cursor location (user might invoke synthesis at any location within the range of edit). We use the same experimental parameters as Section 5.6.3.

Evaluation Results. Table 5.3 shows our evaluation result. Scenarios presents the number of scenarios. In each scenario, one set of history versions and one cursor location are provided to the engine. Our evaluation results show that our engine only generates one false positive and three false negatives on all the scenarios. In other words, we achieve 99.66% precision and 98.98% recall.

Meanwhile, BluePencil$_{cur}$ should be fast enough to ensure that the suggestion can be generated at run-time. Therefore, we also evaluate the efficiency of

BluePencil$_{\text{cur}}$ by measuring the time to generate each suggestion. Time describes the averaged time to generate edit suggestions. Our engine produces one suggestion in 51.8ms on average, and up to 441ms. At the cursor location, we believe generating suggestions in less than 0.44 seconds is acceptable.

> Given one set of history versions and one cursor location, BluePencil$_{\text{cur}}$ achieves around 99% precision and recall in generating correct suggestions. Meanwhile, it takes 51.8 milliseconds on average to generate one suggestion.

5.6.5 A Comparison to BluePencil

In this section, we present an experiment that simulates a developer performing repetitive edits in two different settings.

- *Setting 1*: The developer uses BluePencil to complete the task.
- *Setting 2*: BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$, which are built on top of BluePencil, are enabled and they assist the developer to complete the task.

The goal of this experiment is to compare the amount of information, in the form of examples and locations, that a developer must provide to complete a task when supported by these tools.

Experimental Setup. To simulate Setting 1, given an edit session that contains edits $\{e_1, e_2, ..., e_n\}$, we iteratively add each edit e_i as an example to BluePencil. At each iteration, we check the suggestions produced by BluePencil. If it produces a suggestion to automate an edit e_j, such that $j > i$, we remove this edit from the set of available edits. At the end of the simulation, we have the total number of examples #examples provided by the developer and the number of edits #suggestions that were automated by BluePencil. For instance, consider the scenario showed in Figure 5.1, where the developer performed seven repetitive edits. After providing e_1 and e_2 (Figure 5.1a) as examples to BluePencil, it produces the suggestions to automate e_3 and e_4 (Figure 5.1b). The three edits left are the ones that were applied to the locations shown in Figure 5.1c. We provide e_5 and it produces a suggestion to e_6. Finally, we provide e_7, the last edit. In total, we simulated the developer

Table 5.4: Summary of the comparison to BluePencil. Column #inferredLocs is the number of additional inputs that are automatically identified by the feedback system. Column %automated shows the percentage of edits automated by the synthesis engine.

Approach	Edit	#examples	#locs	#inferredLocs	#suggestions	%automated	Time
Setting 1	350	191	-	-	159	45%	0.25s
Setting 2	350	87	87	37	263	75%	0.32s

providing 4 examples (i.e., #examples = 4) and the BluePencil automating 3 edits (i.e., #suggestions).

Setting 2 is similar to Setting 1 but instead of simulating the developer interaction just with BluePencil, we add BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$. Now, at each iteration after the first, before providing e_i, we first provide an arbitrary cursor location within the location of e_i. Only if BluePencil$_{\text{cur}}$ cannot produce the suggestion to automate e_i, we provide the full example. This process simulates a developer first navigating to the location of e_i and then performing the edit. If BluePencil$_{\text{cur}}$ is able to produce a suggestion for e_i as soon as the developer navigates to the location of e_i, it is counted towards the number of locations #locs. Otherwise, the developer has to manually perform this edit, and e_i is counted towards the number of examples #examples.

Further, we also enable BluePencil$_{\text{auto}}$ to automatically find additional inputs. For instance, back to our running scenario, after providing the first edit in Figure 5.1a as an example, we provide a cursor location within the second edit. Using this example and location, BluePencil$_{\text{cur}}$ produces suggestions for e_2, e_3, and e_4. Additionally, BluePencil$_{\text{auto}}$ produces suggestions for e_5, e_6, and e_7. Note that BluePencil$_{\text{auto}}$ requires at least two examples (see Section 5.5), and thus will not produce any suggestions until the user provides at least one edit and one cursor location. In this simulation, the developer provided one example (i.e., #examples = 1) and one cursor location (i.e., #locs = 1 and the system automated 6 edits (#suggestions = 6). Further, since 3 edits (e_5, e_6, and e_7) were automated using BluePencil$_{\text{auto}}$, we say that these locations are *automatically inferred* and write #inferredLocs = 3. We use the same experimental parameters as Section 5.6.3.

Evaluation Results. Table 5.4 shows the results of our simulation. In Setting 1, BluePencil required 191 examples and produced suggestions for 159 out of 350

edits *i.e.*, the synthesis engine assisted the developer to automate 45% of the edits. Meanwhile, in Setting 2, the synthesis engine automated 263 edits, which represents 75% of the total number of edits. It required only 87 examples and 87 cursor locations. Additionally, BluePencil$_{\text{auto}}$ found 37 additional inputs, decreasing the number of cursor locations the developer has to provide.

While Setting 2 (BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$) is more effective at producing suggestions, the tool should also be fast enough to ensure that the suggestions can be generated at run-time when developers are programming. Therefore, we also evaluated its efficiency by measuring the time to generate edit suggestions. Column Time displays the averaged time to generate edit suggestions. Our engine produced suggestions in 0.32 seconds on average, fast enough to be used as an on-the-fly synthesizer in an IDE. Compared to BluePencil, it was slightly slower as it continuously refines the transformation by invoking the synthesis engine multiple times.

> In Setting 2 (BluePencil$_{\text{cur}}$ and BluePencil$_{\text{auto}}$), the synthesis engine automated 75% of the edits, compared to 45% edits automated in Setting 1 (BluePencil). On average, our engine took 0.32 seconds to produce suggestions.

5.6.6 Discussion

In the above experiments, our technique produced a small number of false positives and false negatives. Besides false positives and negatives related to the limitations of Refazer itself, we found false positives related to semi-supervised synthesis and the automated feedback oracles. We also observed false positives related to the limitations of our anti-unification algorithm.

The semi-supervised synthesis technique produces a false positive in the following case. Given edit: `Model(..., outputs: null, inputs: null)` $\mapsto$ `Model(..., outputs: null)`, i.e., removing `inputs: null`, and the additional positive input: `Model(..., inputs: new List<ModelInput>(), outputs: null)`, semi-supervised synthesis generates a transformation that deletes the last argument. (Note that the order of the last two parameters has been reversed.) Therefore, the synthesized transformation will produce the suggestion for the additional input by

deleting the last argument `outputs: null`. However, the desired edit is deleting the `inputs` : ∗ clause, which is the second last argument in the additional input. That is, the correct suggestion should be to remove the second-to-last argument.

One way to address this issue would be to extend the anti-unification algorithm to handle commutativity as the order of "`name: value`" style arguments is irrelevant. However, this would complicate our anti-unification problem, with having to handle standard arguments under the AU (associativity and unity) theory and the named arguments under the ACU (associativity, commutativity, and unity) theory.

Limitations of the Feedback Oracles. In our experiment, BluePencil$_{\mathsf{cur}}$ and BluePencil$_{\mathsf{auto}}$ produced false positives and negatives due to limitations in the feedback oracles. It might classify negative inputs as positive ones if the locations are too similar. For instance, developers made the following edit: `comparedEdge.Item2 >= Source.Index` ↦ `comparedEdge.Item2 > Source.Index`. The developer's intention was to change `>=` to `>` only if the left side of the comparison expression was `comparedEdge.Item2`. The oracle classified `comparedEdge.Item1 >= Source.Index` as a positive addition since the input is very similar. As future work, we plan to allow users to provide feedback about false positives, so that the system can create negative inputs. On the other hand, the false negatives mainly happened on small inputs where the change was on the root of the AST. In this case, any generalization of the input looked like an over generalization for the feedback oracle, since there was not much context for transformation.

Threats to Validity. Our benchmark suite may not be representative of the different types of edits developers perform. To reduce this threat, we collected real-world scenarios from developers who are working on different large code-bases to have as much variety as possible in the benchmark suite. Another threat is that developers may perform irrelevant, non-repetitive edits in addition to the repetitive ones, which may affect the effectiveness of our technique. To alleviate this issue, we also collected the traces of irrelevant edits and used them in our benchmarks. Finally, in some scenarios of repetitive edits, it is difficult even for humans to discern the transformation intended by the developer, which may affect the construction of our benchmark. To reduce this threat, we manually reviewed these ambiguous scenarios. Wherever possible, we contacted the developer for confirmation.

Chapter 6

Alleviate Overfitting Using Output-Oriented Synthesis

Semi-supervised synthesis learns transformation programs (rules) based on input-output examples and additional inputs. The additional inputs, which are a set of inputs without available correct outputs, can help disambiguate how to generalize the transformation rule by providing more examples of input ASTs. Apart from additional inputs, we observe that there are a large number of available additional outputs. The additional outputs are the after-transformation codes without before-transformation codes being available. This leads to our research question: *whether the additional outputs can also be used to synthesize transformation rules?*

6.1 Introduction

In this chapter, we propose a novel *output-oriented program synthesis* to generate program transformation rules according to (1) a set of input-output edit examples, and (2) a set of additional outputs. The additional outputs are embedded with the human intelligence which demonstrates the structures of the after-transformation code. The additional outputs can be helpful in synthesizing transformation rules in the following two aspects: (1) help us disambiguate how to generalize the transformation rule; (2) help synthesize more transformation rules by providing after-transformation AST patterns. The synthesis goal is to produce a transformation rule that is consistent with the given input-output edit examples, and the synthesized transformation rule should be able to produce the additional outputs on some "unknown" inputs. We achieve this using the following workflow in our technique to synthesize transformation rules:

- Since there could be a large number of additional outputs, our technique first determines the additional outputs that are useful for the synthesis task. An additional output will be regarded as useful if it can help improve the transformation rule. Basically, the given input-output edit examples represent some code structures before- and after-transformation. If one additional output reflects after-transformation code structures that are not reflected in the input-output examples, we regard it as a useful additional output since it could be helpful for synthesizing more substantial transformation rules.

- For each useful additional output, our technique then infers its corresponding input by analyzing its relationship with the given examples. Linking the inferred input with the additional output constructs an additional example, which can be then used as a normal input-output example by any existing synthesizer.

- Since the generated additional examples represent new code structures, they may not be unified with existing input-output examples to produce a single transformation rule. Therefore, our technique groups the examples, including the user-provided and inferred examples, into clusters, and then synthesizes a transformation rule for each cluster. Determining which synthesized transformation should be applied to a given input depends on the context of the code that should be transformed.

We then use the output-oriented program synthesis to automate API usage adaptations. In the process of software development, developers usually rely on third-party libraries to implement certain functionalities. To enable developers to use different components, these libraries usually provide a set of public Application Programming Interfaces (APIs), which define the contract of using the libraries, such as the kinds of calls that can be invoked, the ways to invoke them, the right arguments that should be passed, etc. The client applications that rely on a certain library must use the API correctly and respect the contract built by the APIs. However, when library evolves to accommodate new features or fix security vulnerabilities, it may change the contract defined via APIs and cause its existing client applications to break. The changes that can fail client applications are called *breaking changes*, which makes around 15% of API modifications [139].

Fixing API usage errors caused by breaking changes is a time-consuming and error-prone task. In order to use up-to-date libraries, the developers/maintainers of clients have to keep track of the library update, analyze the changed code, and manually fix the API usage errors. Due to the complexity, developers are not willing to update their dependencies. Indeed, 82% of developers prefer to keep using outdated libraries [56]. The practice that uses outdated vulnerable libraries will expose the clients to the risk of malicious attacks. This becomes more serious with financial applications (e.g., bank clients) which could cause a bigger impact. This indicates the necessity and importance of automatically updating clients' dependencies in an efficient manner.

In recent years, we have seen an emerging trend of tools and techniques that synthesize abstract transformation rules using examples of human code edits and apply the synthesized rules to automate program transformations [82, 81, 109, 7, 87]. Existing program transformation techniques have been studied to automatically update clients' dependencies [92, 20, 41, 26, 143, 40]. Those techniques first infer transformation rules from the before- and after-adaptation examples from human-adapted clients, and then apply the inferred rules to adapt the clients that are relying on outdated libraries. Existing approaches infer transformation rule by generalizing the concrete human adaptations in different ways. For instance, Meditor [143] and AppEvolve [26] simply generate the most general rule by abstracting all the project-specific details (e.g., variable identifiers), which may lead to over-generalized transformation rules. Many approaches, e.g., LASE [82] and ReFazer [109], synthesize the most specific rule over the given examples. To synthesize a properly generalized rule, these approaches require multiple examples. However, multiple human adaptation examples are not always available in reality because client developers are not willing to upgrade their dependencies. Further, for a library that is updated recently, there could be very few clients that have been adapted to the new library. Even though CocciEvolve [40] learns from a single adaptation example, it can only adapt Android deprecated-API usages by introducing an if-condition.

The output-oriented program synthesis is suitable to address the above limitations. Although the number of available human adaptations is limited, the new clients usually use the updated library directly, which gives us an opportunity to mine

usages for the new version(s) of the libraries. In this setting, output-oriented program synthesis takes human adaptations as input-output examples and **the usages of the updated library** as additional outputs to synthesize transformation rules. Relying on output-oriented program synthesis to synthesize transformation rules has two main advantages. First, with the help of additional output, our technique does not require a large number of available examples to synthesize a proper transformation rule. Second, mining usage of new libraries is much more efficient than mining adaptation examples since we just need to search usages in the latest version of the client instead of going through all the commits.

We realized our approach in a tool, called APIFix, on top of ReFazer and evaluated APIFix on a benchmark with seven well-known C# libraries and 138,206 clients that depend on those libraries. Totally, we collect 218 human adaptations (concrete examples) and 2973 new usages (additional outputs) and evaluate our approach in three experiments. First, we measure the effectiveness of output-oriented program synthesis via cross-validation. Evaluation results show that output-oriented program synthesis achieves 91% accuracy (the ground truth manually computed on the human adaptation in client codes) in correctly transforming programs. Second, we applied APIFix on 2154 API usages of outdated libraries, achieving 98.7% precision and 91.5% recall. Last, we compared with output-oriented program synthesis with existing program synthesis tools ReFazer and semi-supervised program synthesis. Evaluation results show that our approach improves both precision and recall over ReFazer. Compared with semi-supervised program synthesis, our technique significantly improves the precision, while not affecting the recall significantly.

6.2 Motivating Example

In this section, we give a high-level overview of the output-oriented program synthesis in automating API usage adaptations by presenting an example from DbUp. DbUp is a .NET library that helps developers to deploy changes to SQL Server databases. It supports most of the widely-used databases, such as MySQL, SQLite, SQLServer, Oracle, etc. DbUp has 10.7M total downloads according to the Nuget Statistics and more than 1800 open-source dependents according to the dependency

graph of Github[1]. At the time of this thesis's writing, DbUp has officially released 40 versions ranging from v1.0.8 to v4.5.0. Each release, especially the major releases, may change some public APIs and hence introduced a number of breaking changes to the old versions. For instance, when DbUp was updated from v3.3.5 to v4.0 [2], the constructor of a widely used class `SqlScriptExecutor` was changed as follows:

```
- public SqlScriptExecutor(Func<IConnectionManager>, Func<IUpgradeLog>,
-    string, Func<bool>, IEnumerable<IScriptPreprocessor>) ...
+ public SqlScriptExecutor(Func<IConnectionManager>, Func<IUpgradeLog>,
+    string, Func<bool>, IEnumerable<IScriptPreprocessor>, Func<IJournal>)...
```

When client applications upgrade their dependency DbUp from v3.3.5 or older versions to v4.0, they may receive compilation error "*'SqlScriptExecutor' does not contain a constructor that takes 5 arguments*". Even though the change of this constructor is simply inserting an additional parameter, it is not easy for client developers to figure out what new argument should be passed, how the new argument is relevant to the other arguments, and how the surrounding context affects the creation of the new argument. In order to use the latest version of the library (i.e., DbUp 4.0), the client developers have to read documents of the library, understand the change, and manually fix the usage errors, which is an error-prone and time-consuming task.

Fortunately, the DbUp developers have provided several examples on how to perform the adaptation within the DbUp codebase itself. For instance, the test cases of the *SqlScriptExecutor*'s constructor are also updated along with this breaking change. Figure 6.1 shows three example edits that are relevant to the `SqlScriptExecutor` constructor. Basically, developers modified the `SqlScriptExecutor` object creations by inserting an additional argument (`() => Substitute.For <IJournal>()`) to match the new constructor signature in DbUp v4.0. Meanwhile, we observe that there are still 411 clients relying on DbUp v3.3.5 or even older versions. Out of which, 84 clients use the `SqlScriptExecutor` constructor which require the similar adaptations. Figure 6.2 shows three `SqlScriptExecutor` object creation examples relying on the DbUp v3.3.5 that require adaptations.

Several approaches have been proposed to help developers to update their API

[1]https://github.com/DbUp/DbUp/network/dependents
[2]https://github.com/DbUp/DbUp/compare/3.3.5...4.0.0-beta0003#diff-bfbdbc9a

```
E1: - new SqlScriptExecutor(() => new TestConnectionManager(dbConnection,
    true),() => new ConsoleUpgradeLog(), null, () => true, null)
    + new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),()
        => new ConsoleUpgradeLog(), null, () => true, null, () =>
        Substitute.For<IJournal>())

E2: - new SqlScriptExecutor(() => new TestConnectionManager(dbConnection,
    true),() => new ConsoleUpgradeLog(), "foo",() => true, null)
    + new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),()
        => new ConsoleUpgradeLog(), "foo",() => true, null,() =>
        Substitute.For<IJournal>())

E3: - new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true)
        {IsScriptOutputLogged = true}, () => new ConsoleUpgradeLog(),
        "foo", () => true, null)
    + new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true)
        {IsScriptOutputLogged = true}, () => new ConsoleUpgradeLog(),
        "foo", () => true, null,() => Substitute.For<IJournal>())
```

Figure 6.1: History edits on *SqlScriptExecutor* that adapt clients from DbUp v3.3.5 or older version to v4.0.

```
I1: new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),()
    => new ConsoleUpgradeLog(), null,() => false, null)
    ▷ new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),()
        => new ConsoleUpgradeLog(), null,() => false, null, () =>
        Substitute.For<IJournal>())

I2: new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true) {
    IsScriptOutputLogged = true },() => new ConsoleUpgradeLog(), "foo",() => true,
     null)
    ▷ new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true)
        IsScriptOutputLogged = true ,() => new ConsoleUpgradeLog(), "foo",() =>
        true, null, () => Substitute.For<IJournal>())

I3: new SqlScriptExecutor(()=> c.ConnectionManager,() => c.Log, schema,() => c.
    VariablesEnabled, c.ScriptPreprocessors)
    ▷ new SqlScriptExecutor(()=> c.ConnectionManager,() => c.Log, schema,() =>
        c.VariablesEnabled, c.ScriptPreprocessors, () => c.Journal)
```

Figure 6.2: Code from clients that still use DbUp v3.3.5 or older versions

usages by learning a transformation rule from the human edits (Figure 6.1) [92, 20, 41, 26, 143], and automatically transform the code requiring adaptations (Figure 6.2). For example, ReFazer [109] learns a transformation rule R by looking at the edit history, and represents the learned rule using a domain-specific language (DSL). The DSL will be explained in Section 6.3.2. For simplicity, we show the rule R for this

example as follows:

$$new\ SqlScriptExecutor(X_1, X_2, X_3, X_4, X_5) \mapsto$$
$$new\ SqlScriptExecutor(X_1, X_2, X_3, X_4, X_5, ()=>Substitute.For<IJournal>())$$
$$\textbf{where}\ \ X_1.\texttt{Type} = \text{“}\texttt{Func}\text{”}$$
$$X_2.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_2.\texttt{Text} = \text{“}() => newConsoleUpgradeLog()\text{”}$$
$$X_3.\texttt{Type} = \text{“}\texttt{String}\text{”}$$
$$X_4.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_4.\texttt{Text} = \text{“}() => true\text{”}$$
$$X_5.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_5.\texttt{Text} = \text{“}null\text{”}$$

Terms X_1, X_2, X_3, X_4, X_5 represent the least general generalization of the five arguments in the three examples, respectively. Each term is guarded by predicates in terms of `Type` and `Text`. Term X_1 and X_3 do not have a `Text` predicate because the text of the first and third arguments in the given examples are different. The existing inductive program synthesis techniques [109, 82] prefer to synthesize the least general generalization across the examples to avoid false positives. Unfortunately, R is not applicable to I_1 and I_3 because the text of X_2 is not “`() => new ConsoleUpgradeLog()`”, X_4 is not “`() => true`” or X_5 is not “`null`”. Therefore, R cannot transform I_1 and I_3, hence produces false negatives. In contrast, if we simply generalize R by ignoring all predicates on `Text` (i.e., delete predicate on X_2.`Text`, X_4.`Text` and X_5.`Text`), the generalized R will be applicable to all the usages in Figure 6.2. However, it may produce false positives, i.e., transform some code in an incorrect way. For instance, the most likely correct transformation of I_3 should be inserting `() => c.Journal` as the last argument (it is not clear what is the correct output since the ground truth is not available, and the most likely correct transformation of I_3 is shown in ▷ red in Figure 6.2). However, the inferred transformation rule from existing examples inserts `() =>Substitute.For<IJournal>()`, which is very likely to be a false positive. How to balance false negatives and false positives is one of the main challenges in the transformation rule inference.

Our solution: To address the above challenges, we propose to learn transformation rules not only from edit examples but also from the usages of new library versions. The main insight behind our idea is that the clients created after the release

```
O1: new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),()
    => logger, null,() => true, null,() => Substitute.For<IJournal>())

O2: new SqlScriptExecutor(() => Substitute.For<IConnectionManager>(),() => null,
    null,() => false, null,() => Substitute.For<IJournal>())

O3: new SqlScriptExecutor(() => c.ConnectionManager,() => c.Log, schema,() => c.
    VariablesEnabled, c.ScriptPreprocessors, () => c.Journal)

O4: new SqlScriptExecutor(() => connectionManager,() => Substitute.For<
    IUpgradeLog>(), null,() => true, null,() => versionTracker)
```

Figure 6.3: Code from clients that use DbUp v4.0 or newer versions

of DbUp v4.0 are very likely to use the latest version. We find many examples from those clients that use DbUp v4.0, which are also embedded with human intelligence on how to use the updated *SqlScriptExecutor* constructor. From these usages of DbUp 4.0, we can learn the API usage patterns and infer transformation rules. These examples can improve our transformation rule to support different types of code structures as explained above. Our solution would mine usages of the new version of the library DbUp 4.0, by automatically crawling through Github repositories based on the dependency graph of the library. In total, we find 30 usages of `SqlScriptExecutor` relying on DbUp v4.0 in existing clients, and Figure 6.3 shows four of them. By referring to the first usage O_1, we would learn that the inferred transformation rule R is also applicable even if X_2.`Text` (*() => logger* in O_1) is not exactly the same as the given edit examples (*() => new ConsoleUpgradeLog()*). Similarly, by looking at O_2, we would know that R can be applied even if X_4.`Text` is *() => false*. This knowledge helps us to improve the transformation rule R learned from the edit examples E_1, E_2, and E_3 by generalizing the context where R can be applied. Furthermore, using O_3 and O_4, we can learn that the inserted last argument is not necessarily `()=>Substitute.For<IJournal>()`. Inspired by O_3, we can infer

a new transformation rule R_1. The simplified representation of R_1 is as follows:

$$
\begin{aligned}
&\mathit{new}\ \mathit{SqlScriptExecutor}(X_1, X_2, X_3, X_4, X_5) \mapsto \\
&\mathit{new}\ \mathit{SqlScriptExecutor}(X_1, X_2, X_3, X_4, X_5, () => X_6.\mathit{Journal}) \\
&\quad \textbf{where}\ X_1.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_2.\texttt{Text} = \text{“}() => c.\mathit{ConnectionManager}\text{”} \\
&\qquad\qquad X_2.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_2.\texttt{Text} = \text{“}() => c.\mathit{Log}\text{”} \\
&\qquad\qquad X_3.\texttt{Type} = \text{“}\texttt{String}\text{”} \wedge X_2.\texttt{Text} = \text{“}\mathit{schema}\text{”} \\
&\qquad\qquad X_4.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_4.\texttt{Text} = \text{“}() => c.\mathit{VariablesEnabled}\text{”} \\
&\qquad\qquad X_5.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_5.\texttt{Text} = \text{“}c.\mathit{ScriptPreprocessors}\text{”} \\
&\qquad\qquad X_6.\texttt{Type} = \text{“}\texttt{UpgradeConfiguration}\text{”} \wedge X_6.\texttt{Text} = \text{“}c\text{”}
\end{aligned}
$$

Similarly, we will also infer another transformation rule R_2 from O_4. Rules R, R_1 and R_2 form a complete disjunctive transformation rule. Which rule R, R_1 or R_2 should be applied is determined according to the context, i.e., the values of $X_1 \ldots X_5$ in this case. This additional knowledge helps us infer substantial transformation rules. With the additional outputs, we can be more confident about how to generalize the rule or create new rules, and hence be more confident on the transformed codes by the synthesized rules. Combining the knowledge learned from human edits from Figure 6.1 and usages of new library from Figure 6.3, our technique can automatically adapt all the client codes shown in Figure 6.2 to DbUp v4.0. After transforming the usages in Figure 6.2, we manually verified that all the transformed codes can be successfully compiled.

6.3 Output-Oriented Program Synthesis

In this section, we first introduce the output-oriented program synthesis problem and present the technical details of our solution to this problem.

6.3.1 Problem Statement

Semi-supervised synthesis assumes the availability of additional inputs, however, finding additional inputs is an error-prone task. If provided with additional inputs that should not be manipulated (i.e., invalid additional inputs), semi-supervised synthesis will generate over-generalized transformation rules.

Example 6.3.1 *Let us revisit the example shown in Section 6.2. Suppose we take the human edits in Figure 6.1 as input-output examples E, and the old usages from Figure 6.2 as additional inputs AI, semi-supervised synthesis will synthesize a transformation rule that is applicable to all the inputs from E and all the additional inputs I_1, I_2 and I_3 from Figure 6.2. Therefore, it will over-generalize the predicates on $X_1 \dots X_5$ (the arguments of* `SqlScriptExecutor`*'s constructor), and hence transform I_3 by incorrectly inserting* () => `Substitute.For<IJournal>()`.

Instead of using additional inputs for the synthesis, we use additional outputs in the synthesis process. Compared to additional inputs, considering additional outputs has two main advantages. First, determining whether an additional input is valid for the synthesis task requires human feedback. Although we proposed semi-automated and fully automated feedback in Chapter 5.4 to generate additional inputs, they are implemented based on heuristic hence may not always work. Synthesizing with invalid additional inputs can lead to over-generalized or over-specified transformation rules. In contrast, the additional outputs are the after-transformation codes written by developers, which are guaranteed to be valid additional outputs for the to-be-synthesized transformation rule. Compared with additional inputs, using additional outputs does not require the involvement of human in the synthesis process. Second, the additional outputs have been embedded with human intelligence, e.g., the structure of the after-transformation codes from which we could learn new transformation rules that are not reflected in the given input-output examples.

Formally, the output-oriented program synthesis takes a set of input-output examples $\{i_0 \mapsto o_0, \dots, i_n \mapsto o_n\}$ and a set of additional outputs $\{ao_0, \dots, ao_m\}$. The synthesis goal is to produce a transformation rule R such that $R(i_k) = o_k$ for $k \in 0 \dots n$ and $R(ai_j) = ao_j$ for $j \in 0 \dots m$, where ai_j represents some input.

6.3.2 Domain-Specific Language

State-of-the-art program synthesis tools, like ReFazer [109], search for a transformation rule that satisfies the provided examples over a predefined Domain-Specific Language (DSL) (refer to Chapter 2.2.2). The output-oriented program synthesis inherits the DSL of ReFazer, and extends it to the language $\mathcal{L}$ shown in Figure 6.4 (the differences are highlighted in grey).

```
rules       := rule | Disjunction(rules, rule)
rule        := (guard, transformer)
guard       := pred | Conjunction(pred, guard)
pred        := IsNthChild(node, n) | IsKind(node, kind)
               | Attribute(node, attr) = value | Not(pred)
               | IsType(node, type)
node        := Path(input, path)
transformer := construct | select
construct   := Tree(kind, attrs, children)
children    := EmptyChildren | Cons(node, children)
               | InsertChild(Children(select), pos, node)
               | DeleteChild(Children(select), pos)
               | ReplaceChildren(Children(select), posList, children)
               | MapChildren(λ input: transformer, Children(select))
select      := Nth(Filter(guard, SubTrees(input)), n)
pos         := n | ChildIndexOf(node)

Variables:
  AST input;  List<int> posList;   string kind, attr, value;
  int n;      XPath path;          Dictionary<string, string> attrs;
```

Figure 6.4: Domain-specific language for program transformation rule

Additional outputs are embedded with human intelligence that can help to synthesize new transformation rules, as discussed in Section 6.3.1. The transformations reflected by additional outputs and input-output examples may not be able to be unified to a single transformation rule.

Example 6.3.2 *Consider input-output examples* E_1*:* `handler.Handle(request)` $\mapsto$ `handler .Handle(request, token)` *and* E_2*:* `TestSubject.Handle (request)` $\mapsto$ `TestSubject.Handle (request, token)`*, and additional output AO:* `_handler .Handle(request, new CancellationToken())`*. By looking at* E_1 *and* E_2*, the inferred transformation rule is to insert a second argument* `token`*, while by looking at AO, we know the inserted argument can be* `new CancellationToken()`*. The transformation inferred from the additional output cannot be unified with the transformation rule synthesized from the given examples, because the inserted code elements are different and cannot be unified together.*

Different from traditional synthesis techniques that produce a single transformation rule in the form of (`guard`, `transformer`), the output-oriented program synthesis

can generate multiple transformation rules $\{(\texttt{guard}_0, \texttt{trans}_0), \ldots, (\texttt{guard}_n, \texttt{trans}_n)\}$. The above example will produce two transformation rules that insert different codes as the second argument. The *transformation rules* are defined by a set of disjunctive transformation rules, where each of $\texttt{trans}_i$ applies to a different interval in the domain defined by $\texttt{guard}_i$. Hence, applying the synthesized transformation rules to a given AST node *node*, we have:

$$\texttt{if } (\texttt{guard}_0(\mathit{node})) \{ \texttt{ return } \texttt{trans}_0(\mathit{node}) \} \texttt{ if } \ldots$$
$$\texttt{if } (\texttt{guard}_n(\mathit{node})) \{ \texttt{ return } \texttt{trans}_n(\mathit{node}) \} \texttt{ return } \bot$$

Note that, the `guard` of transformation rules can overlap, i.e., for a AST node *node*, there may exist multiple `guard`s such that $\texttt{guard}(\mathit{node}) = \texttt{true}$. In this situation, *node* can be transformed in multiple ways.

Further, different from semi-supervised synthesise and its predecessor, our approach uses typed ASTs. A typed AST associates each node with a set of attributes including the node kind (e.g., Identifier, Expression, etc), the node type (e.g., Integer, Boolean, etc), the text value (source code fragment corresponding to the node), and a set of child nodes. For a node that does not have a type, we leave its type empty. We use `SubTree(`T`)` to denote the set of sub-trees of T. We use $\mathbb{T}$ to represent the set of all ASTs.

6.3.3 Output-Oriented Program Synthesis

In this section, we present the technical details of output-oriented program synthesis. The procedure is depicted in Algorithm 8 and works as follows:

- Given a set of input-output examples E and additional outputs AO, we first determine which additional outputs are useful for improving the synthesized transformation rule (lines 1 - 2);
- For each useful additional output, we infer a candidate input, and hence create a set of additional examples AE (lines 12 - 19).
- We then categorize the given input-output examples E and inferred additional examples AE into clusters, and synthesize transformation rules via $\textsc{ReFazer}_{\texttt{guard}}$ and $\textsc{ReFazer}_{\texttt{transformer}}$ (lines 4 - 9). $\textsc{ReFazer}_{\texttt{guard}}$ takes all the inputs from

Algorithm 8: Output-oriented program synthesis

Input: Input-output examples: $E = \{i_0 \mapsto o_0, \ldots, i_n \mapsto o_n\}$,
additional output: $AO = \{ao_0, \ldots, ao_m\}$
Output: re-write rule: R

$(\tau, <\sigma_0, \ldots, \sigma_n>) := \bowtie \{o_0, \ldots, o_n\}$;
$selectedAO := \{ao \mid \neg \texttt{IsInstance}(ao, \tau) \wedge \ ao \in AO\}$;
$AE := \texttt{InferExample}(selectedAO, E)$;
$EditClusters := \texttt{ClusterEdit}(E \cup AE)$;
$R = \{\ \}$;
for $editCluster \in EditClusters$ **do**
  $R_{guard} := \textsc{ReFazer}_{\texttt{guard}}(\{i \mid (i \mapsto o) \in editCluster\})$;
  $R_{transformer} := \textsc{ReFazer}_{\texttt{transformer}}(editCluster)$;
  $R := R \cup (R_{guard}, R_{transformer})$;
end
return R;

Function `InferExample`(AO, E)**:**
  $AE := \{\ \}$;
  $\pi := \texttt{Provenance}(i_0 \mapsto o_0)$;
  for $ao \in AO$ **do**
    $(\tau, <\sigma_0, \sigma_1>) := o_0 \bowtie_\pi ao$;
    $AE := AE \cup \{\ \sigma_1(\sigma_0^{-1}(i_0)) \mapsto ao\ \}$;
  end
  return AE;

input-output examples, and produces a `guard` that is true on all of them, while $\textsc{ReFazer}_{\texttt{transformer}}$ takes a set of examples and produces a `transformer` consistent with them.

Filtering additional outputs Our objective is to find the patterns reflected in the additional outputs AO but not reflected in the given input-output examples E. Program synthesis is a generalization process of the given concrete input-output examples. Informally, program synthesis infers a generalized transformation rule $\tau_i \mapsto \tau_o$ such that i_k is a instance of τ_i and o_k is an instance of τ_o for $k \in \{0, \ldots, n\}$. If an additional output ao is not an instance of τ_o (like a counter-example), it is a concrete output that cannot be generated by the synthesized transformation rule. Therefore, ao has the potential to improve the transformation rule in the synthesis process.

To find useful additional outputs, the output-oriented program synthesis first generates a common pattern for all the outputs in E via anti-unification technique [98] (line 1). Given a set of ASTs $\{o_0, \ldots, o_n\}$, anti-unification process (denoted by $\bowtie \{o_0, \ldots, o_n\}$) produces a pair ($\tau$, $<\sigma_0, \ldots, \sigma_n>$), where τ is a generalized AST with labelled holes $\{h_0, \ldots, h_l\}$, and $\sigma_0, \ldots, \sigma_n : \{h_0, \ldots, h_l\} \mapsto$ `AST` are a set of substitutions, such that $\sigma_0(\tau) = o_0 \ldots$ and $\sigma_n(\tau) = o_n$. The anti-unification process produces the most specific generalization τ of the given ASTs. For each additional output ao, output-oriented program synthesis then checks whether ao is an instance of τ (line 2) by searching for a substitution σ=$\{h_0 \mapsto subtree_0, \ldots, h_l \mapsto subtree_l\}$, where $subtree_j \in$ `SubTrees`(ao) for $j \in 0 \ldots l$, such that $\sigma(\tau) = ao$. If a substitution σ exists, ao is an instance of τ. Otherwise, ao is not an instance of τ, and it will be regarded as a useful additional output that will be utilized in the following synthesis steps.

Example 6.3.3 *Let us revisit the two input-output examples shown in Example 6.3.2. Suppose we have two additional outputs*

AO_1 : `this.inner.Handle(request, token)`

AO_2 : `_handler.Handle(request, new CancellationToken())`

Anti-unifying the outputs of the two given examples `handler.Handle(request, token)` $\bowtie$ `TestSubject .Handle(request, token)` *generates (*h_0`.Handle(request, token)`*,* $<\{h_0 \mapsto$ `handler`$\}$, $\{h_0 \mapsto$ `TestSubject`$\}>$*). Because we can find a substitution* σ=$\{h_0 \mapsto$ `this.inner`$\}$*, such that* σ(`Handle(request, token)`)=AO_1*, we will not regard* AO_1 *as an useful additional output. However,* AO_2 *is a useful additional output since we cannot find such a substitution.*

Additional input inference To utilize the additional outputs (AO) in the synthesis process, our key idea is to infer a set of additional input-output examples using the provided additional outputs by analyzing its relation with E. Specifically, for each ao in AO, we infer a corresponding input ai to create an additional example $ai \mapsto ao$ (line 17). To obtain this additional example, ao is first anti-unified with an output from E, (e.g., o_0) using *anti-unification modulo provenance* as explained in Chapter 5.4. Provenance analysis calculates which fragments of the outputs are

dependant on which fragments of the inputs. Anti-unification modulo provenance of o_0 and ao, denoted as $o_0 \bowtie_\pi ao$, produces an generalization $(\tau, <\sigma_0, \sigma_1>)$ by just anti-unifying the provenance nodes.

Example 6.3.4 *Consider the input-output example* `handler.Handle(request)` $\mapsto$ `handler.Handle(request, token)`*, the provenance analysis would produce four nodes {*`handler.Handle`*,* `handler`*,* `Handle`*,* `request`*}, since these nodes in the output can be constructed using the nodes from input. Anti-unification modulo provenance of* `handler.Handle(request, token)` *and* `_handler.Handle (request, new CancellationToken())` *just unifies* `handler` *with* `_handler`*, since* `handler` *is a provenance node. However,* `token` *and* `new CancellationToken()` *will not be unified since* `token` *is not a provenance node.*

Anti-unification modulo provenance produces σ_0 and σ_1, representing the substitutions that are applied to o_0 and ao, respectively. Typically, $\sigma_0(\tau) = o_0$ and $\sigma_1(\tau) = ao$, for generating additional input, we apply the same substitution to the origin input $\sigma_1(\sigma_0^{-1}(i_0))$ (line 17). These additional examples can be used to expand the given input-output examples for the synthesis process. This allows our approach to be integrated with any existing program synthesis technique.

Example 6.3.5 *Following Example 6.3.4, the two generated substitutions are* $\sigma_0 = \{h_0 \mapsto$ `handler`$\}$ *and* $\sigma_1 = \{h_0 \mapsto$ `_handler`$\}$*. Next, by applying* $\sigma_1\sigma_0^{-1}$ *to* `handler.Handle(request)`*, it would generate* `_handler.Handle(request)`*, which is the corresponding inferred input for the additional output.*

Synthesis procedure Given the input-output examples E and the inferred additional examples AE, output-oriented program synthesis then synthesizes a set of transformation rules. In the synthesis procedure, AE can be helpful in the following two aspects: 1) *guard generalization*: determine the proper context where the transformation rule should be applied; and 2) *transformation rule enhancement*: enhance the transformation rules by encoding more substantial transformation operations. The transformation operations from both E and AE may not be able to be unified in a single transformation rule as discussed in Section 6.3.2. Therefore, examples in $E \cup AE$ are first classified into clusters according to their transformation operations

(line 4 ClusterEdit). Specifically, for each input-output example $i \mapsto o$, we calculate a set of edit operations $\{op_0, \ldots, op_n\}$ that can transform input i to output o using GumTree [25]. Just as the edit script in GumTree, the edit operations include `Insertion`, `Deletion` and `Update`. The input-output examples with the same edit operations are grouped into the same cluster. Basically, each cluster represents a distinct transformation operation. Second, for each cluster, we utilize ReFazer$_{\texttt{guard}}$ and ReFazer$_{\texttt{transformer}}$ to generate a transformation rule R_{guard} and $R_{transformer}$, respectively (lines 7 - 9). The output-oriented program synthesis generates a more properly generalized guard since it takes into account the inputs of both E and AE, and it synthesizes more substantial transformation rules by considering the different transformations from different clusters. The transformation rule of each cluster is combined together to form the complete set of transformation rules.

Example 6.3.6 *Consider the input-output example E:* `handler.Handle(request)` $\mapsto$ `handler.Handle(request, token)` *and the additional example AE:* `_handler.Handle(request)` $\mapsto$ `_handler.Handle(request, new CancellationToken())`. *The edit operation of E is {* `INSERT("token")` *}, while the edit operation of AE is {* `INSERT("new CancellationToken()")` *}. Since the edit operations of E and AE are different, they will be categorized into different clusters.*

6.4 APIfix: Automated API Usage Adaptation

In this section, we present how output-oriented program synthesis is used to automate API usage adaptations. To achieve this, output-oriented program synthesis first synthesizes transformation rules using human-adapted examples (input-output examples) and the usages of the updated library (additional outputs), and then applies the synthesized transformation rules to all codes that require a transformation. Considering usages of the updated library in the synthesis process enables us to learn substantial API usage patterns of the new library.

Figure 6.5 depicts the architecture of output-oriented program synthesis for automating API usage adaptations. Given a library and its clients, APIfix first determines the breaking changes caused by library update. For each broken API, the *Miner* of APIfix mines relevant *human adaptations*, *new usages* and *old usages*

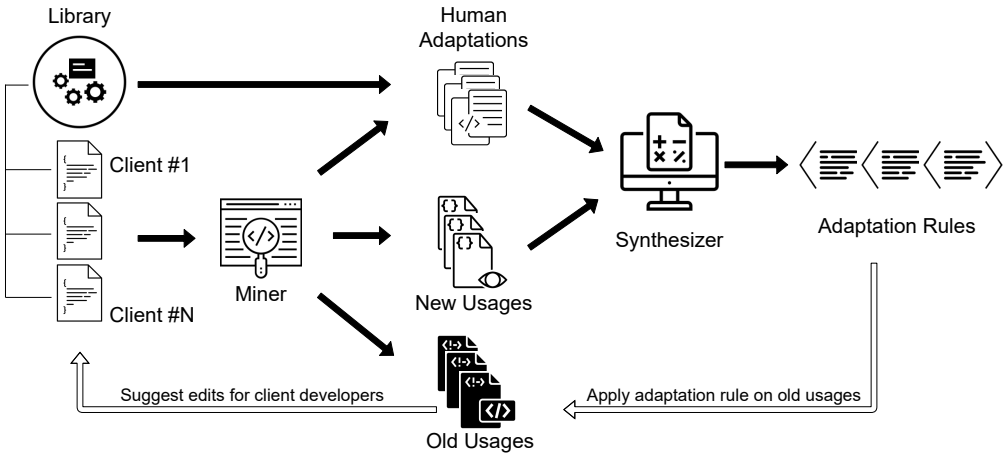


Figure 6.5: APIFIX: output-oriented program synthesis for automating API usage adaptations

from both library itself and client codes. Specifically, the human adaptations are extracted from the library itself and the clients that have already been adapted to the new library version by client developers. Note that mining human adaptations from clients is time-consuming, it is an optional step in APIFIX. The *new usages* and *old usages* are mined from clients, which represent the usages of the old and new library version, respectively. The output-oriented program synthesis takes as inputs the human adaptations and new usages, and synthesizes a set of transformation rules. APIFIX then applies the synthesized transformation rules to transform the old usages and applies the transformed code back to clients for verifying syntactic correctness. The syntactically correct code edits are then sent to developers as edit suggestions. We will present the technical details of each step in the following.

6.4.1 Mining Human API Usage Adaptations and Library Usages

Given a library, the first step to automate API usage adaptations is to determine which public APIs have been modified in a library update that leads to *breaking changes*, mine the human adaptations of such API usages, and mine the usages of the new library version.

Change summary Given a library X with two versions v_o and v_n, representing the old and new library version, we first construct a *change summary* by comparing v_o

and v_n. Basically, change summary models the set of modified APIs introduced by the X's update from v_o to v_n, including the modified identifier names, modified modifiers, modified function/constructor arguments, inserted/deleted public classes/methods, and so on. The workflow to generate change summary is as follows:

- For each file f_n in X's version v_n, we use the Git version control system to determine the corresponding file f_o in X's version v_o;
- If f_n is different from f_o, then, for each API api_n in f_n, we use clone detection to determine its matched api_o in f_o that is most similar to api_n;
- If the signatures of matched $<api_n, api_o>$ are different, we will save the signatures of $<api_n, api_o>$ into the change summary.

Human adaptations Given the change summary of library X's update $v_o \rightarrow v_n$, we then mine the human adaptations that adapt codes relying on v_o to the new library version v_n. The human adaptations can be mined from two sources: the library X itself and the clients that depend on library X. When the developer of X introduces a breaking change on an API, he or she also needs to adapt the usages of the relevant API within X (e.g. tests of API). The human adaptations within X itself are great sources to learn transformation rules because (1) those adaptations are available immediately after the new library version is released; and (2) the developers of library X know best about the breaking changes. Besides X itself, human adaptations can also be mined from clients of X that have already upgraded their dependency on X from v_o to v_n. For each client, we need to traverse through all commit history and determine the commit(s) that upgraded the dependency on X from v_o to v_n. For a modified API $<api_n, api_o>$ modeled in the change summary, we mine its corresponding human adaptations by searching for the usages of api_o in the client code before upgrading X, and the corresponding usages of api_n in the client code after upgrading X. The usages of api_n and api_o form human adaptations.

API usages Similar to finding human adaptations for the APIs, the usages of the API are also mined from its open-source clients. Instead of traversing through all the commits, mining API usages just requires checking the latest version of clients.

Algorithm 9: APIFix: Automated API Usage Adaptation

Input: Human adaptation: E; Old Usages: OU; New Usages: NU; Similarity Thresholds: T1, T2
Output: Transformed Old Usages: TOU

1 $EditClusters$:= `ClusterEdit`(E) ;
2 TOU := { };
3 **for** $editCluster \in EditClusters$ **do**
4 $\quad i_0 \mapsto o_0$:= `GetFirst`($editCluster$) ;
5 $\quad \pi$:= `Provenance`($i_0 \mapsto o_0$);
6 $\quad relevantNU := \{nu \mid nu \in NU \wedge$ `Distance`$(o_0, nu, \pi) < T_1\}$;
7 $\quad relevantOU := \{ou \mid ou \in OU \wedge$ `Distance`$(i_0, ou, \pi) < T_2\}$;
8 $\quad rules$:= `synthesiser`($editClsuter, relevantNU$);
9 $\quad$ **for** $ou \in relevantOU$ **do**
10 $\quad\quad TOU := TOU \cup \{ou \mapsto t \mid t \in rules(ou)\}$;
11 $\quad$ **end**
12 **end**
13 **return** TOU;

14 **Function** `Distance`(t_1, t_2, π)**:**
15 $\quad (\tau, <\sigma_1, \sigma_2>) := t_1 \bowtie_\pi t_2$;
16 $\quad$ **return** `TreeDistance`$(\sigma_1^{-1}(t_1), \sigma_2^{-1}(t_2))$;

For a clients that relying on v_n, we mine the usages of api_n to form *new usages*, while for a clients that relying on v_o, we create *old usages* by mining the usages of api_o.

6.4.2 Clustering Algorithm

Given a set of human adaptations E, a set of new usages NU and old usages OU of a certain API, Algorithm 9 depicts the workflow of automated API usage adaptations. First, since there could be multiple adaptation strategies for a certain broken API, APIFix categorizes the human adaptations into different clusters based on the adaptation operations (line 1). The motivation behind this categorization is that the edits in the same cluster should have the same adaptation strategies which can be represented by a single transformation rule. We use the same clustering algorithm (i.e. `ClusterEdit`) used in Section 6.3.3. For each cluster, APIFix then determines the new usages that are relevant to the edits in this cluster according to their similarity (line 6). Typically, if a new usage is similar to the output of the edits in this cluster, it will be regarded as a relevant new usage. The similarity is defined

as the tree distance of two ASTs (lines 15-16). However, instead of calculating the tree distance of the two concrete ASTs, APIFix first abstracts them using the anti-unification modulo provenance, because directly comparing the concrete ASTs will result in false negatives.

Example 6.4.1 *Consider the following example for a breaking change which requires a new argument to the function call,* `handler.Handle(request)` $\mapsto$`handler.Handle(request, token)` *and a relevant new usage* `handler.Handle(new Request{Value = pValue}, token)`*, the tree distance between the new usage with output of the given example is large because* `request` *and* `new Request{Value = pValue}` *are quite different. However, the anti-unification module provenance tells us that* `request` *is a relevant part of input since it also appears in the output. Therefore,* `request` *and* `new Request{Value = pValue}` *can be abstracted because we just care about the high-level API usage patterns. By comparing the abstracted nodes, APIFix will determine this new usage is relevant to this cluster.*

Similarly, we also determine the relevant old usages according to their similarity with the inputs of the edits in this cluster. These old usages are the codes that should be transformed by the transformation rules learned from this cluster.

6.4.3 Synthesizing and Applying Transformation Rule

Given the edits in a cluster and the corresponding relevant new usages, we invoke the output-oriented program synthesis to produce transformations (line 8), i.e., *transformation rules* in the context of API usage adaptation. The synthesized transformation rules are then applied to transform the relevant old usages. Recall that, APIFix synthesizes a set of transformations (rules) $\{(\texttt{guard}_0, \texttt{trans}_0), \ldots, (\texttt{guard}_\texttt{n}, \texttt{trans}_\texttt{n})\}$. When applying the rules to an old usage *ou*, if there exist multiple `guard`s such that $\texttt{guard}(ou) = \texttt{true}$, we could generate multiple $\texttt{trans}(ou)$. In this situation, we will try to apply each $\texttt{trans}(ou)$ to the client code and check whether it causes compilation errors. We save all transformations that do not cause compilation errors.

6.5 Evaluation

In this section, we evaluate the output-oriented program synthesis and answer the following research questions:

RQ1 What is the effectiveness of output-oriented program synthesis in generating correct code transformations?

We split the mined human adaptations into training and testing sets. We evaluate the effectiveness of output-oriented program synthesis via cross-validation by measuring the syntactic and semantic equivalence between auto-transformed codes with the human adaptations.

RQ2 How does APIfix perform in automating API usage adaptations?

APIFix should generate effective suggestions for API usage adaptation that can be used by developers. We measure the number of false positives and false negatives generated by APIFix.

RQ3 How does APIfix compare with the state-of-the-art techniques?

Our main contribution is to enable the program synthesis system to utilize the additional output. We measure the output-oriented program synthesis by comparing it with the general program synthesis ReFazer and semi-supervised program synthesis.

Implementation APIFix is implemented in Python and C#, and it is composed of three main components: `Miner`, `Build Engine` and `Synthesis Engine`. The Miner, which is implemented using GitHub APIs, is used to mine GitHub repositories to find breaking changes, existing human adaptations, and library usages. The Build Engine is used initially to build typed ASTs and finally to validate the transformed codes. We implemented the Build Engine on top of Microsoft MSBuild [86], and used Roslyn framework [111] to parse source files and generated ASTs. The Synthesis Engine is implemented on top of an extended ReFazer [106] (it is extended to support our DSL).

Dataset To evaluate our output-oriented program synthesis, we build our dataset by mining from GitHub repositories. Our miner searches for the "Most starred" C# libraries, and select libraries to construct our dataset using the following criteria:

- The library has at least 300 dependents reported in the statistics of the GitHub Dependency graph [37];
- The library has multiple released versions, and there is at least one breaking change;
- There are available human adaptations that adapt library/clients to the new library version;
- There are new usages and old usages of the broken APIs.

Finally, we select seven libraries with 138,206 clients. From those libraries/clients, we mined 218 human adaptations, 2973 new usages and 2154 old usages following the procedure described in Section 6.4.1. The detailed statistics of the selected dataset are shown in Table 6.1. Column **"#Clients"** presents the number of clients for each library. For each library, column **"API Name"** gives the modified APIs that are broken by the library update from **"Old version"** to **"New version"**. As we mentioned in Section 6.4.1, the human adaptations can be mined from the library itself and its clients. Column **"#Edit$_l$"** and **"#Edit$_c$"** show the number of human adaptations mined from library itself and clients, respectively. The last two columns present the number of new usages and old usages, respectively. For simplicity, the number of new/old usages is bounded to 1,000.

In our experiment, we set the threshold T_1 and T_2 in Algorithm 9 as 0.25 and 0.15, respectively. All experiments are conducted on a Dell Precision Tower 7810 with Intel(R) Xeon(R) CPU E5-2630 processor and 32GB RAM running 64-bit Windows 10.

6.5.1 Exp-1: Effectiveness of the Output-Oriented Program Synthesis

We first evaluate the effectiveness of our output-oriented program synthesis using a cross-validation experiment on our dataset. We use the human adaptations

Table 6.1: Statistics on our dataset used for evaluation

Library	#Clients	Old version	New version	API name	#Edit$_l$	#Edit$_c$	#New usages	#Old usages
Polly	8531	5.5.0	6.1.2	Execute	61	11	3	13
		6.1.2	7.0.0	WrapAsync	3	3	19	18
		6.1.2	7.0.0	WaitAndRetryAsync	3	5	30	195
MediatR	14099	5.0.1	6.0.0	Handle	1	6	721	56
		6.0.0	7.0.0	Process	1	0	18	33
DbUp	1819	3.3.5	4.0.0	StoreExecutedScript	2	0	28	147
		3.3.5	4.0.0	SqlScriptExecutor	6	0	36	84
		3.3.5	4.0.0	AdHocSqlRunner	2	0	4	28
SteamKit	332	2.0	2.1	Disconnect	2	0	1	38
AutoMapper	89151	7.0.0	8.0.0	Ignore	2	30	38	220
		7.0.0	8.0.0	ResolveUsing	16	42	1000	271
FluentValidation	20568	8.0.0	9.0.0	Validate	9	6	1000	1000
MimeKit	3716	1.22.0	2.0.0	DecodeTo	1	6	75	51
Total	138,206	-	-	-	109	109	2973	2154

for each breaking change collected in our data-set to measure the accuracy of the automatically transformed codes by comparing them to the developer transformed codes.

Experimental Setup For each subject, we take human adaptations from the library itself (column "`Edit`$_l$") as the training set to synthesize the transformation rules, and human adaptations from clients (column "`Edit`$_c$") as testing set to validate the correctness of the synthesized transformation rules. Specifically, output-oriented program synthesis generates transformation rules by taking the human adaptations from the library as input-output examples and the new usages (column "`#New usages`") as additional outputs. The synthesized transformation rules are then evaluated on the human adaptations from clients. We measure the correctness of automatically transformed codes by manually checking their syntactic and semantic equivalence with human-adapted codes, i.e., the ground truth.

Experimental Results The results of our experiment are summarized in Table 6.2. Columns "Library" and 'API name' indicate the names of the library and API name that introduces a breaking change, respectively. Column "#Instances" indicate the number of transformations for each API for which the ground truth

Table 6.2: Exp-1: Cross-validation results of output-oriented program synthesis

Library	API name	#Instances	Syntactic	Semantic	Accuracy
Polly	Execute	11	0	9	81%
	WrapAsync	3	1	3	100%
	WaitAndRetryAsync	5	2	3	60%
MediatR	Handle	6	0	6	100%
AutoMapper	Ignore	30	30	30	100%
	ResolveUsing	42	42	42	100%
FluentValidation	Validate	6	6	6	100%
MimeKit	DecodeTo	6	0	0	0%
Total	-	109	81	99	91%

is available. Columns "Syntactic" and "Semantic" represents the number of transformations for which the result is syntactically and semantically equivalent to the ground truth, respectively. Column "Accuracy" shows the percentage of correct (syntactically or semantically) transformations with respect to the total number of instances for each breaking change.

In total, out of 109 instances, 81 transformed codes by output-oriented program synthesis are syntactically equivalent to human-adapted codes, while 99 of them are semantically equivalent. Output-oriented program synthesis achieves 91% overall accuracy in correctly transforming old usages of the APIs in client codes. In our evaluation, we noticed there can be multiple possible ways to transform an old usage in client code. For instance, the human adaptation transformed the API invocation `requestHandler.Handle(request)` to two statements `var token = new CancellationToken(); requestHandler.Handle(request, token)` [3]. In contrast, our technique transforms it to `requestHandler.Handle(request, new CancellationToken())`, which is syntactically different, but semantically equivalent to the human adapted code. On the other hand, our technique generates 10 false negatives. The main reason is that the synthesized transformation rule is over-specialized to the given examples and additional outputs.

Example 6.5.1 *Consider the following example, a synthesized transformation rule*

[3]https://github.com/transformania/tt-game/commit/a58e410

from given human adaptation and new usages is simplified as follows:

$$\texttt{Policy } X_1 = X_2.\texttt{WaitAndRetryAsync}(\dots) \mapsto$$
$$\texttt{AsyncPolicy } X_1 = X_2.\texttt{WaitAndRetryAsync}(\dots)$$

This transformation rule fails to transform

$$\texttt{RetryPolicy retry} = \texttt{polly.WaitAndRetryAsync}(\dots)$$

which should be transformed to

$$\texttt{AsyncRetryPolicy retry} = \texttt{polly.WaitAndRetryAsync}(\dots)$$

. The reason is that that synthesized rule is not general enough to be applied to this case and hence produces a false negative.

RQ1: In a cross-validation experiment, the output-oriented program synthesis achieves an overall **91%** accuracy, indicating its effectiveness in synthesizing correct transformation rules.

6.5.2 Exp-2: Effectiveness in Automating API Usage Adaptations

We apply our output-oriented program synthesis to automatically adapt transformations and generate patches to assist client developers to upgrade their library usages from the old version to the new version. We evaluate the effectiveness of APIFix in automating these usage adaptations, and generating valid transformations which can be directly applied by the client developers.

Experimental Setup In this experiment, we take the human adaptations from the library itself ("$\texttt{Edit}_l$") as input-output examples, and new usages as additional outputs. We only use "$\texttt{Edit}_l$" as our input-output examples because mining human adaptations from clients can be a time-consuming task in practice. Mining human adaptations from the library are much faster than from clients, which enables APIFix to quickly synthesize transformation rules and generate code edit suggestions. The synthesized transformation rules are then used to transform the old usages. We evaluate the correctness of the transformed codes as follows:

Table 6.3: Evaluation results of APIFix with different synthesis techniques. APIFix represents our tool with Output-Oriented Program Synthesis. APIFix^{R} and APIFix^{S} represent APIFix with ReFazer and Semi-Supervised Program Synthesis as the synthesis engine, respectively. APIFix^{O+S} uses a combined Semi-Supervised and Output-Oriented Program Synthesis as the synthesis engine.

API	APIFix			APIFix^{R}			APIFix^{S}			APIFix^{O+S}		
	TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP
Execute	8	0	0	7	1	0	8	0	5	8	0	0
WrapAsync	2	2	0	4	0	0	4	0	2	2	2	0
WaitAndRetryAsync	2	2	0	2	2	0	4	0	0	4	0	0
StoreExecutedScript	120	24	3	24	123	0	123	0	24	120	24	3
SqlScriptExecutor	84	0	0	40	28	16	56	8	20	84	0	0
AdHocSqlRunner	28	0	0	14	14	0	28	0	0	28	0	0
Disconnect	38	0	0	12	0	26	12	0	26	38	0	0
Ignore	220	0	0	220	0	0	220	0	0	220	0	0
ResolveUsing	271	0	0	271	0	0	271	0	0	271	0	0
Handle	54	0	2	0	0	56	35	0	21	54	0	2
Process	14	0	0	3	11	0	3	11	19	14	0	0
Validate	7	0	6	0	13	0	13	0	8	7	0	6
DecodeTo	0	51	0	0	51	0	0	0	51	0	0	51
Total	848	79	11	597	243	98	777	19	176	850	26	62

- Upgrade the dependency version for each client which should cause compilation error(s);
- Apply the transformed code to the client;
- Check whether the transformed code fixes the compilation error(s) caused by the target broken APIs.

We only evaluate the syntactic correctness of the transformed codes because the correct code transformation (i.e., ground truth) is not available for the clients that still rely on the old version of the library. Checking the semantic correctness of the transformed codes can be left for the developers. Recall that APIFix is designed to *assist developers* (e.g. providing code edit suggestions) instead of replacing developers.

Experimental Results In Table 6.3, columns 3-5 summarize the evaluation results. Columns `TP`, `FN` and `FP` represent the number of true positives, false negatives

and false positives, respectively. Although we apply the synthesized transformation rule to all the old usages, not all of them need to be transformed. For instance, the breaking change of `WaitAndRetryAsync` is changing its return type from `Policy` to `AsyncPolicy`. The old usage `var policy = polly.WaitAndRetryAsync(...)` does not need to be transformed, since the implicit "type" `var` allows compiler determines its real type at compilation time. That is, `var` allows the type `polly` to be `Policy` at old version, and to be `AsyncPolicy` at the new version, hence this statement does not need to be transformed. Among the old usages requiring transformation, APIFix produces 848 true positives, 79 false negatives, and only 11 false positives, achieving 98.7% precision and 91.5% recall. Similar to the result shown in Exp-1, APIFix produces false negatives since the synthesized transformation rule is over-specialized to the given data. It produces false positives mainly because it transforms some codes that should not be transformed.

RQ2: By learning from human adaptation and new usages, APIFix automatically adapts 848 old usages with **98.7%** precision and **91.5%** recall.

6.5.3 Exp-3: Comparison with State-of-The-Art Technique

We provide an empirical evaluation for APIFix by comparing it with the recently proposed program transformation techniques. Specifically, we consider three different comparable synthesis techniques: original ReFazer (APIFix^{R}), semi-supervised program synthesis (APIFix^{S}) and a combination of semi-supervised and output-oriented program synthesis (APIFix^{O+S}). We provide the same mining procedure to all the considered approaches to make a fair comparison, and we only replace the synthesis technique.

Experimental Setup Given input-output examples E, additional input AI and additional output AO, APIFix^{R} synthesizes transformation rule using E, APIFix^{S} utilizes both E and AI, APIFix uses E and AO, and APIFix^{O+S} uses all E, AI, and AO. Specifically, APIFix^{O+S} combines both semi-supervised and output-oriented program synthesis by constructing additional examples via (1) AE_1: inferring corresponding inputs for additional outputs (APIFix), and (2) AE_2: inferring corresponding outputs for additional inputs (semi-supervised synthesis). If any

Table 6.4: The precision and recall of APIFIX, APIFIX^{R}, APIFIX^{S} and APIFIX^{O+S} in transforming old usages

Approach	APIFIX	APIFIX^{R}	APIFIX^{S}	APIFIX^{O+S}
Precision	98.7%	85.9%	81.5%	93.2%
Recall	91.5%	71.0%	97.6%	97.0%

inferred additional example in AE_1 and AE_2 conflict with each other, we simply drop the conflicted examples in AE_1. Consider the following example from Section 6.2, for a certain input, the inferred examples from AE_1 and AE_2 transform it in different ways.

```
new SqlScriptExecutor(()=>c.ConnectionManager, _, _, _, _, )↦
```

AE$_1$: `newSqlScriptExecutor(()=>c.ConnectionManager, _, _, _, _, ()=>Substitute.For<IJournal>())`

AE$_2$: `newSqlScriptExecutor(()=>c.ConnectionManager, _, _, _, _, ()=>c.IJournal)`

In this scenario, we drop the example from AE_1 that causes the conflict because the outputs of AE_2 are produced by developers which should be given higher confidence. For this experiment, we use the same setting as Exp-2 described in Section 6.5.2. We only evaluate the syntactic correctness of the transformed codes because the correct code transformation (i.e., ground truth) is not available for the clients that still rely on the old version of the library.

Experimental Results The summarized results of our comparison with the state-of-the-art techniques are also shown in Table 6.3. Columns `TP`, `FN` and `FP` represent the number of true positives, false negatives and false positives, respectively for each tool. Table 6.4 presents the precision and recall of each tool. Compared with APIFIX^{R}, APIFIX significantly reduces both numbers in false negatives and false positives. This is because considering additional output enables APIFIX to synthesize more accurately generalized transformation rules. However, compared with APIFIX^{S}, APIFIX performs better with fewer false positives but incur more false negatives. Considering additional inputs enables APIFIX^{S} to generalize the synthesized rule to the additional inputs. Therefore, APIFIX^{S} produces fewer false negatives.

Example 6.5.2 *Let us revisit the false negative in Example 6.5.1 produced by our ap-*

proach again. If the input `RetryPolicy retry = polly.WaitAndRetryAsync(...)` *is regarded as additional input that should be transformed, the transformation rule will be further generalized as*

$$\texttt{X}_3\ X_1 = X_2.\texttt{WaitAndRetryAsync}(\dots) \mapsto$$
$$\texttt{AsyncPolicy}\ X_1{=}X_2.\texttt{WaitAndRetryAsync}(\dots)$$

where X_3 *represents an identifier which can be matched with* `RetryPolicy`*. With the generalized rule, the target input is transformed to* `AsyncPolicy retry = polly.WaitAndRetryAsync(...)`*. Even though the transformed code is syntactically different from human adaptation, since* `AsyncPolicy` *is the parent class of* `AsyncRetryPolicy`*, they are semantically equivalent.*

However, if APIFIX^S over-generalizes the synthesized rule to the inputs that should not be transformed, it will lead to false positives. Example 6.3.1 presents such a false positive. Balancing false positives and false negatives is challenging.

Furthermore, we propose to combine both semi-supervised and output-oriented program synthesis, i.e., APIFIX^{O+S}, to utilize both additional inputs and additional outputs. Experimental results show that combining additional input and additional output can further improve the number of true positives. However, the improvement is not significant. Our results are based on a straightforward and simple combination of semi-supervised and output-oriented program synthesis. We believe other possible combinations may further improve the results. How to better combine them to infer better transformation rules can be an interesting question as future work.

RQ3: APIFIX improves both precision and recall over APIFIX^R. Compared with semi-supervised program synthesis APIFIX^S, our technique significantly improves the precision, while does not significantly affect the recall.

6.5.4 Threats to Validity

Three threats may affect the validity of our empirical evaluation. First, in our experiments Exp-2 (Section 6.5.2) and Exp-3 (Section 6.5.3), we evaluated the correctness of transformed code using the compiler. The compiler can ensure syntactic correctness, but it cannot check semantic errors. To solve this problem, we

performed a cross-validation in Exp-1(Section 6.2). The evaluation results in Exp-1 and Exp-2 are consistent. Second, although APIFix shows its effectiveness on the evaluated benchmark, it may not perform well on other subjects. To mitigate this problem, we selected a fairly large dataset with more than 2500 cases that cover different scenarios. Last, we manually compare automatically transformed code with human adaptations to verify their correctness. In case of the potential bias caused by manual analysis, I invited my labmate to independently checked the correctness of the transformed code.

Limitation One of the limitations of APIFix is that it only considers the API usages without dependency analysis. When the usage of an API is updated, its dependent statements may also need to be modified accordingly. For instance, suppose the return type of an API invocation is changed, the statements that use the returned value should also be modified. To support such systematic changes, it is necessary to perform program dependency analysis, which is out of the scope of this thesis. Existing API usage adaptation techniques, such as LibSync [92], have investigated this problem. APIFix can be potentially integrated with the program dependency analysis employed by those techniques in the future.

Chapter 7

Related Work

The contributions of this thesis are related to several areas of research: program repair, program synthesis, test case generation, program transformation, API usage adaptation, and so on.

7.1 Automated Program Repair

Automated program repair techniques take in a buggy program, and a set of specifications, and aim to generate a patched program satisfying the given specifications. In this section, we briefly discuss the program repair techniques. For a general summarization of program repair techniques, the readers could refer to the overview articles [65, 112] or the surveys [89, 36].

Test-based automated program repair treats the provided test suite as the specification of intended behavior and generates patches that make program pass all the given tests. Typically, patch generation methods include: (1)*search-based approaches*, (2)*constraint solving based approaches*, (3)*Static-based approaches*. While these approaches are able to generate high-quality patches according to the provided tests, the weakness of test suites remains a challenging problem in test-based program repair. Due to the incompleteness of test suites, the generated patches may overfit the available tests and can break untested functionality [121].

7.1.1 Search-Based Program Repair

Search-based approaches first generate a patch space and then search correct patch via heuristic [64, 68, 126], random search [103], test-equivalence analysis [75] or learning approaches [70]. The correctness of patches is validated using the given test suites. Search-based program repair is able to generate high-quality patches and

can easily scale to large programs. However, because of the fact that search-based program repair depends on incomplete test suites, it suffers from overfitting issues. Fix2Fit proposed in Chapter 3 falls in the category of search-based program repair. To alleviate the overfitting problem, Fix2Fit automatically generates more tests to augment the given test suite.

7.1.2 Semantics-Based Program Repair

Semantics-based techniques like SemFix [93], Nopol [144], DirectFix [78], Angelix [79] and JFIX [61] generate patches in two steps. First, they formulate the requirement to pass all given tests as constraints for the identified program statements. Second, they synthesize a patch for these statements based on the inferred constraints. This type of approach is related to ExtractFix because these approaches also involve constraint extraction and patch synthesis. The main difference is that the constraints extracted by semantics-based techniques just represent the specifications to pass the given tests. Therefore, an incomplete test suite will result in incomplete constraints and hence incomplete specifications. In contrast, the constraints extracted by ExtractFix represent the underlying cause of the crash and the conditions that should be satisfied to fix vulnerabilities. Therefore, ExtractFix can alleviate the overfitting problem in automated program repair by generating patches that generalize beyond the given tests. Essentially, ExtractFix only needs a single exploit trace to generalize the vulnerability, where existing semantic repair techniques usually need a test-suite.

Furthermore, Prophet [70], SPR [68] and F1X [75] are somewhere between search based and semantic based approach. Specifically, Prophet first generates a patch search space and ranks the candidate patches according to semantic models learned from successful patches. SPR first searches for the possible values that can pass the given test and synthesizes a patch that can generate such values. F1X enumerates the patch space by evaluating the patches that produce the same values together. All these approaches combine the search-based approach and semantics analysis (e.g., the values produced by patches) with the objective of improving the patch validation efficiency and producing more correct patches. Fix2Fit is orthogonal to those approaches. They can be combined in the following way: Prophet, SPR, and

F1X are used to efficiently generate initial candidate patches (FIX2FIT is actually built on top of F1X), and the test generation proposed in FIX2FIT is used to rule out the overfitted candidates.

7.1.3 Learning-Based Program Repair

Another line of repair is to learn repair strategies from human patches, such as Genesis [67], GetaFix [7] and Phoenix [9]. Those techniques first mine human patches that fix defects in existing software repositories, learn a general code transformation rule, and apply the transformation rule to the buggy programs to produce patches. The repairability of those techniques does not rely on predefined transformation operators. Instead, they can automatically learn repair strategies from available human patches. However, because of the fact that they learn repair strategies across different projects, they can only fix common bugs, e.g., null-pointer dereference. Different from them, the techniques proposed in the thesis focus on the specification issues, which can be a complementary technique to the learning-based repair systems. Our approaches can be potentially combined with those techniques to improve the patch quality generated by them.

7.1.4 Static Program Repair

Instead of relying on test cases, several approaches propose program repair driven by static analysis and verification techniques. These approaches generate patches for static analysis violation by reasoning in separation logic [128] or learning repair strategies from the wild [9]. Specifically, the work of [128] generates patches that are guaranteed to satisfy certain heap properties (this covers few common bug types such as memory leaks, resource leaks, or null pointer dereference). Their approach is still search-based, where semantic search [54] is used to identify code snippets that satisfy the desired properties. Furthermore, the entire framework is based on the reasoning in separation logic and is used to fix heap properties only. Differently, our approach is not limited to any specific kind of bugs and it can program synthesis to generate a patch.

Vulnerability Repair The recent work SENX [44] aims to repair vulnerabilities using a combination of predicate generation, patch placement, and patch synthesis.

The main difference with SENX is that SENX does not have any analytical understanding of which fix locations are suitable and what fixes to insert, and usually inserts trivial if-conditions to disable the crash at/near the crash location ([44] Table III). Besides, SENX does *not* perform any constraint propagation. In the absence of constraint propagation, SENX relies on heuristics to guide patch generation, which limits it to specific classes of bugs. In contrast, our approach is not limited to certain vulnerabilities. Most of the patches generated by our tool are more general and modify expressions/statements different from the crash location.

7.2 Alleviate Overfitting in Program Repair

In this section, we discuss the approaches that alleviate the overfitting problem in program repair.

Test Generation. Automatically generating more tests for automated program repair is a useful strategy to alleviate the overfitting problem. Existing approaches generate additional test cases using symbolic execution [117], grey box fuzzing [146] (like AFL), or evolutionary algorithm [147] (like EvoSuite [30]). All those approaches are relevant to Fix2Fit since they are also designed to generate tests for APR system. Their goal is to cover the patched methods or statements, but they do not take the patch semantics into consideration. DiffTGen [140], the work most relevant to Fix2Fit, generates test inputs that exercise syntactic differences, monitors execution results, and then selects tests that uncover differences between the original faulty program and the patched program. Compared with DiffTGen where the patch is validated one by one, Fix2Fit is more efficient since it examines the patches in the same patch partition together. Besides, different from all existing approaches, Fix2Fit utilizes the semantic difference between patches as a search heuristic and guides the test case generation process, so that we can efficiently find more behavioral differences across patches. Another main difference lies in inferring the expected behaviors (oracles) for newly generated test inputs. Existing approaches infer oracles of tests based on test similarity [141], developers' feedback [140, 116] or some obvious oracles (like memory safety [146]). In contrast, Fix2Fit utilizes security oracles from sanitizers to avoid introducing crashes or vulnerabilities. Similar to sanitizers, Valgrind's *memcheck* is utilized by CodePhage [117] to filter out patches that cause

overflows. However, CodePhage can only support memory related errors (e.g., (invalid reads and writes; uninitialized reads and writes), while Fix2Fit can support more classes of errors by using different sanitizers.

Patch Ranking. One way of addressing overfitting in program repair is to rank patches according to statistical information learned from code repositories [80]. Typical approaches learn from existing patches [70, 62, 113], existing source code [142], or both [49, 138] to rank the patches in the order of likelihood to be correct. On the other hand, Xiong, et al. [141] propose to filter out the patches based on syntactic and semantic distance between patched and original program. ClearView [95] ranks and even discards patches by executing the patched program for a while (e.g., ten seconds) in real deployment environment. Since these approaches are based on statistical information or heuristics, there is no guarantee that the discarded patches are incorrect patches and the generated patches can be generalized beyond tests. In contrast, Fix2Fit discards patches by generating more tests, which ensures the discarded patches are overfitted patches. Further, ExtractFix extracts crash-free constraints and ensures the constraint is satisfied on all tests.

Reference Implementation. In many development scenarios, there exists a reference implementation, and the developers try to be compatible with the reference implementation while optimizing other aspects such as performance. For example, when implementing a Java compiler, OpenJDK is the reference implementation, and other implementations such as Jikes JVM tries to optimize the performance. Based on this observation, [77] proposes program repair with a reference implementation, where the reference implementation serves as an oracle to avoid overfitting. Compared with this approach, our approaches do not need a reference implementation.

Customized Program Repair. Another line of research to alleviate overfitting problem is to use the search space that is likely to be correct. This strategy is present in essentially all patch generation systems because they only consider a restricted set of patches. Furthermore, many APR systems target specific classes of bugs/errors, such as fixing memory leaks [32, 66], concurrency bugs [14, 51], integer overflows [71], etc. This type of work designs specific candidate patch space for certain types of bugs. By just considering the specifically designed set of patches for each type of bug, these APR systems have a higher chance to generate correct

patches. In contrast, our work does not focus on a specific type of bug but tries to derive a general approach. Besides, one of the most related works with ExtractFix is TAP [118], which eliminates buffer overflows and integer overflow according to predefined templates. There are two main differences between ExtractFix and TAP. First, TAP uses predefined templates to directly generate patches at the crash location, while ExtractFix just relies on predefined templates to generate CFCs and designs a general framework to generate patches according to the inferred CFCs. Therefore, ExtractFix can be easily extended to support other kinds of bugs. Second, TAP just eliminates the observed bugs (exit on buggy program state), in contrast, ExtractFix is designed to patches just like developers.

7.3 Goal-Directed Test Generation

Goal-directed test generation can be used to generate test inputs to maximize code coverage [132, 137], cover the changes in patch [11] or find behavioral asymmetries between programs (differential testing) [97]. Symbolic execution employs constraint collection and solving to systematically and effectively explore the state space of feasible execution paths [13], and can be used for directed testing [73, 114, 101, 96]. In contrast to symbolic execution, grey box fuzzing does not involve heavy machinery of symbolic execution and constraint solving. Greybox fuzzing directs the search to achieve a certain goal by adjusting the mutation strategy according to the information collected at run-time with the help of compile-time instrumentation. Greybox fuzzing has been demonstrated to be useful for increasing code coverage [132, 105], reaching target location [11], and finding behavioral asymmetries between programs [97]. To customize goal-directed test generation for patch evaluation, Fix2Fit takes the semantic of patches into consideration and it is designed with the goal of finding semantic discrepancies between patches.

7.4 Program Synthesis

Program synthesis, while being an old field of study [12, 99, 72], has recently been successfully used in many domains including data manipulation and wrangling [38, 119, 145], data structure manipulation and design [27, 29, 120], concurrent

programming [123, 131, 15, 16], and distributed controller design [129, 3]. The counter-example guided inductive synthesis procedure, that turns any synthesis task into repeated solving of programming-by-example tasks is the basis of the state-of-the-art synthesis technique Sketch [124, 125, 122]. The syntax-guided synthesis (SyGuS) framework [1] attempts to unify synthesis tasks from different domains by providing a mechanism to specify both the syntax and semantics of the desired solution. Efficient general-purpose SyGuS solvers have been built and have found success in various domains [129, 108, 4, 42, 2]. However, general-purpose synthesizers are often less efficient than domain-specific ones as they are not able to leverage domain-specific algorithms and techniques. Further, in most program synthesis techniques, the specification needs to be well-defined and provided explicitly. To avoid the above limitation, our technique, presented in Chapter 5, automatically determines the example specification for the synthesis task. This ability to automatically determine which examples to use allows for the modeless operation of our technique.

7.4.1 Semi-Supervised Program Synthesis

Semi-supervised machine learning techniques [148] combine labeled data (i.e., input-output examples) with unlabeled data (i.e., additional inputs) during training to exploit a large amount of unlabeled data available in many domains, such as websites, source code, and images [149]. Semi-supervised approach can advance machine learning, especially when training data are not sufficient. Beyond classical machine learning settings, semi-supervised learning techniques have also been adapted for use in program synthesis. For example, the BlinkFill system [119] for synthesizing spreadsheet string transformations exploit input data by extracting a graphical constraint system to efficiently encode the logical structure across all available inputs. This input structure allows BlinkFill to achieve a dramatic reduction in the number of candidate programs, leading to improvement in performance and reduction in the number of input-output examples over previous systems [38]. Unfortunately, direct application of this approach to the domain of program transformations is impractical due to different types of inputs (positive inputs and negative inputs), the large number of inputs (all AST nodes in the source code), and the size of the

ASTs themselves (potentially many thousands of tokens per file). To mitigate these issues, we have proposed a novel technique based on reward functions to isolate only those additional inputs that are likely to provide fruitful disambiguation, while still preserving the runtime efficiency required for interactive use in an IDE setting.

7.4.2 Interactive Program Synthesis

Interactive program synthesis systems allow users to incrementally refine the specification in response to synthesizer outputs [59, 5]. Within this paradigm, a notable approach for proposing refinements is based on the concept of distinguishing inputs [48], in which inputs are discovered for which the outputs of multiple consistent programs disagree, suggesting the need for additional refinement to rule out undesired candidate programs. FlashProg [74] employs this notion of distinguishing inputs to pose parsimonious sequences of questions to the user to resolve ambiguities with respect to the user's specification. A disadvantage of this approach, however, is the overhead required for users to answer potentially many rounds of clarifying questions to refine intent. In Chapter 5, we propose a complementary technique: we can synthesize new programs using semi-supervised synthesis. Our approach has the advantage that it allows users to refine intent with little or even no modification to their workflow. Additionally, the technique not only leverages user feedback but also allows fully automated feedback during specification refinement.

7.4.3 Program Synthesis for Software Refactoring

Software refactorings are structured changes to existing software that improve code quality while preserving program semantics [94, 83]. Popular IDEs such as Visual Studio [84], Eclipse [23], and IntelliJ [46] provide built-in support for various forms of well-understood software refactorings. Several program synthesis-based approaches have been studied toward user-friendly refactoring and code transformation support, such as the Sydit, Lase, and ReFazer systems [81, 82, 109] for synthesis of code transformations from examples. Getafix [7] and Revisar [110] apply code mining techniques to discover such changes offline from large codebases, thus expanding breadth while also mitigating the burden for users to specify examples explicitly. BluePencil [87] takes an alternative approach to

increase discoverability and user-friendliness: the system uses a modeless, on-the-fly interaction model in which the programmer is presented with suggested edits without ever exiting the boundaries of the IDE's text editor–the system watches the user's behavior and analyzes code change patterns to discover ad-hoc repetitive edits.

The semi-supervised program synthesis approach is complementary and compatible with the techniques employed by BluePencil: the modeless interaction of BluePencil provides easy discoverability, and additional inputs provide a natural and effective mechanism for refinement when a false negative or positive is discovered.

7.5 Program Transformation

Automated program transformation techniques infer abstracted transformation rules from human-generated transformations and apply the inferred rules to transform the codes in other codebases. Program transformation techniques have been applied in many domains, such as fixing software bug [9, 67, 7], automating repetitive edits [109, 82, 81], and intelligent code refactoring [87, 33]. Similar to our approach, these techniques also learn transformation rules from concrete human transformations. The main difference between our approach with these techniques lies in that our output-oriented program synthesis also considers the human intelligence embedded in the additional outputs and our semi-supervised synthesis also considers the additional inputs. These features reduce the dependency on the human-generated transformations and enable us to learn more substantial transformation rules.

API Usage Adaptation. Existing program transformation techniques have been studied to be applied to automatically update the dependencies of clients [92, 20, 26, 143, 40]. These techniques first infer adaptation rules from the before- and after-adaptation examples from human-adapted clients, and then apply the inferred rules to adapt clients that are relying on outdated libraries. The main difference between these techniques and APIfix is that APIfix synthesized adaptation rules from both input-output examples and additional outputs. Therefore, APIfix can achieve good performance with fewer human adaptations, while these techniques require a large number of human adaptations to synthesize a proper adaptation rule. Even though CocciEvolve [40] learns from a single adaptation example, it can only adapt Android deprecated-API usages by introducing an if-condition. Furthermore, one

of the main focuses of these techniques is to perform program dependency analysis to extract code skeletons before and after API usages, e.g., find the dependent statements of the API usage via data dependency analysis. Differently, the focus of APIFix is the output-oriented program synthesis by just considering the API usages without dependent statements. APIFix is complementary and compatible with the dependency analysis techniques employed by these existing approaches. APIFix can be potentially combined with them to synthesize more complete adaptation rules.

Apart from the inductive adaptation rule inferences, researchers have also studied approaches that are integrated into the development environment with the aim of automatically adapting API usages. For instance, CatchUp [41] records the refactoring actions when developers evolve an API and replays the recorded refactoring actions in the client codes. SemDiff [19] analyzes how a library was adapted to its own changes and then provides adaptation suggestions to client programs. These approaches require that the library and clients are developed in the same development environment. In contrast, the adaptation rule synthesized by APIFix can be applied to any development environment and automatically adapt any client requiring adaptations.

Chapter 8

Conclusion

Programming-by-example techniques inductively construct programs according to specification demonstrated via input-output examples. Since examples are usually incomplete specifications, they prone to overfit the given examples, and hence synthesize incorrect patches/programs. To address the overfitting problem, early techniques utilize existing test generation tools to generate more tests, rely on predefined heuristics to ranks patches/programs, or select patches/programs using learning-based approaches. Although those techniques have shown good performances in many scenarios, they suffer from effectiveness and quality limitations. In this thesis, we propose a series of techniques to alleviate the overfitting issue in automated program repair and program synthesis. All the proposed techniques are unified by the idea of augmenting the specification demonstrated via input-output examples. At its core, we infer developers' intent via program analysis techniques including test generation, semantic reasoning, semi-supervised and output-oriented approaches. The contributions of this thesis are summarised as follows:

8.1 Summary of Contributions

- We propose to integrates fuzzing and automated repair tightly by modifying a fuzzer to prioritize tests that can rule out large segments of the patch space. Results from the continuous fuzzing service OSS-Fuzz from Google show significant promise. (Chapter 3)

- We propose to fix observed vulnerabilities by extracting constraint specifications from an exploit. Even though the vulnerability is observed on a specific test input (the so-called exploit), our extracted constraint captures the "gen-

eral reason" behind the vulnerability via symbolization. By propagating the extracted constraint from the crash location to other potential fix locations, we generate fixes via fix localization and patch synthesis. Our work thus goes beyond test-driven repair and provides a workflow and tool for exploring the fix space of common software security vulnerabilities as well. (Chapter 4)

- We introduce a novel semi-supervised synthesis engine and apply it for predicting repeated edits that exploit the latent information in the user's code. By combining knowledge about what edits the user has performed in the past with the observable patterns in the rest of the code, our technique is able to significantly improve precision and recall metrics for predicting future repeated edits. (Chapter 5)

- We present output-oriented program synthesis and apply it for automated API usage adaptations. Modern software systems heavily depend on third-party libraries. The breaking changes of API caused by library updates can break the client applications. The output-oriented program synthesis technique automatically adapts the client applications to let them use the new version of libraries. Compared with existing program synthesis techniques, output-oriented program synthesis infers transformation rules based on human adaptations (i.e. input-output examples) as well as the usages of the new version of libraries (i.e. additional outputs). The additional outputs can be helpful in finding the correct level of generalization of transformation rules and synthesizing new transformation rules. Evaluation results show output-oriented program synthesis achieves 91% accuracy in correctly adapting API usages. (Chapter 6)

8.2 Perspectives

Current program repair and synthesis techniques are still in the early stage both in research and practice. I believe that this area of research needs more effort from both academia and industry. On one hand, I think dealing with their higher-level design issues needs to be explored, so that they can generate programs efficiently and effectively. On the other hand, I think it is worthwhile to explore practical

scenarios where those techniques can be applied. Specifically, I have the following three potential directions:

- **Other strategies to alleviate overfitting** Although I have investigated a set of approaches to alleviate the overfitting problem via program analysis, I believe there could be many other strategies that can further alleviate the overfitting issues. For instance, the large number of available software repositories can be great sources to help synthesize high-quality programs. The advances of neural network systems can be potentially applied to improve the search efficiency and decide which programs/patches are more likely to be correct. Furthermore, in my thesis work, I have explored synthesizing programs with additional inputs or additional output. However, how to efficiently and effectively combine the additional inputs and additional outputs in program synthesis has not been well studied. I think that this can be an interesting research question to explore in the future. Such a research question can help us gain further insights on reducing *overfitting* (to given input-output examples) in program synthesis.

- **Interactive program repair/synthesis** Even though my thesis work so far focused on solving the overfitting issue and improving the auto-programming techniques, it may still require human to validate the correctness of auto-generated programs (patches). Program repair/synthesis might be integrated into the programming environment to automatically provide suggestions during development/maintenance [35]. This would help to reduce the cost of development/maintenance and also improve software quality.

- **Application of program repair/synthesis** My current research has investigated the possibility of applying auto-programming to software maintenance tasks, e.g., fixing software bugs, code refactoring and API usage adaptations. From a broader perspective, I believe that the idea of program repair/synthesis could be adopted in many domains, such as DNNs, beyond source code engineering. In particular, these techniques could be applied to help normal users (the user without programming experiences) automate some repetitive and boring tasks.

In conclusion, this thesis not only attempts to alleviate the overfitting program in program repair and synthesis via program analysis (i.e., intelligent test generation, semantic reasoning, semi-supervised synthesis, and output-oriented program synthesis), but also opens many opportunities for future research works, including better strategies to alleviate overfitting (e.g., mining software repositories), interactive repair/synthesis, and synthesis to automate repetitive tasks for helping normal users.

Bibliography

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis", in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 1–8.

[2] R. Alur, P. Cerny, and A. Radhakrishna, "Synthesis through unification", in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 163–179.

[3] R. Alur, M. M. K. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing finite-state protocols from scenarios and requirements", in *10th International Haifa Verification Conference*, Springer, 2014, pp. 75–91.

[4] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer", in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2017, pp. 319–336.

[5] S. An, R. Singh, S. Misailovic, and R. Samanta, "Augmented example-based synthesis using relational perturbation properties", *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–24, 2019.

[6] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering", *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[7] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically", *Proceeding of ACM Programming Language*, vol. 3, no. OOPSLA, Oct. 2019.

[8] R. Balzer, "A 15 year perspective on automatic programming", *IEEE Transactions on Software Engineering*, no. 11, pp. 1257–1268, 1985.

[9] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations", in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 613–624.

[10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later",, 2010.

[11] M. Bohme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2017, pp. 2329–2344.

[12] J. R. Buchi and L. H. Landweber, "Solving sequential conditions by finite-state strategies", *Transactions of the American Mathematical Society*, vol. 138, pp. 295–311, 1969.

[13] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs", in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, 2008, pp. 209–224.

[14] Y. Cai and L. Cao, "Fixing deadlocks via lock pre-acquisitions", in *International Conference on Software Engineering (ICSE)*, ACM, 2016, pp. 1109–1120.

[15] P. Cerny, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh, "Quantitative synthesis for concurrent programs", in *23rd International Conference on Computer Aided Verification (CAV)*, Springer, 2011, pp. 243–259.

[16] P. Cerny, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Efficient synthesis for concurrency by semantics-preserving transformations", in *25th International Conference on Computer Aided Verification (CAV)*, Springer, 2013, pp. 951–967.

[17] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: A powerful approach to weakest preconditions", in *Proceedings of the 30th ACM SIGPLAN Confer-*

ence on Programming Language Design and Implementation (PLDI), ACM, 2009, pp. 363–374.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. F. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, pp. 451–490, 1991.

[19] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution", in *2009 IEEE 31st International Conference on Software Engineering*, IEEE, 2009, pp. 599–602.

[20] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–35, 2011.

[21] G. J. Duck and R. H. C. Yap, "Heap Bounds Protection with Low Fat Pointers", in *Proceedings of the 25th International Conference on Compiler Construction*, ACM, 2016, pp. 132–142.

[22] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers." In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] Eclipse Foundation, "Eclipse", At `https://www.eclipse.org/`, 2020.

[24] W. S. Evans, C. W. Fraser, and F. Ma, "Clone detection via structural abstraction", *Software Quality Journal*, vol. 17, no. 4, pp. 309–330, 2009.

[25] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing", in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[26] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps", in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 204–215.

[27] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI*, ACM, 2015, pp. 229–239.

[28] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, "The spirit of ghost code", *Formal Methods in System Design*, vol. 48, no. 3, pp. 152–174, 2016.

[29] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic, "Example-directed synthesis: A type-theoretic interpretation", in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, ACM, 2016, pp. 802–815.

[30] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software", in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (FSE*, ACM, 2011, pp. 416–419.

[31] Z. Fu and S. Malik, "On solving the partial MAX-SAT problem", in *International Conference on Theory and Applications of Satisfiability Testing*, 2006, pp. 252–265.

[32] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for C programs", in *International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 2015, pp. 459–470.

[33] X. Gao, S. Barke, A. Radhakrishna, G. Soares, S. Gulwani, A. Leung, N. Nagappan, and A. Tiwari, "Feedback-driven semi-supervised synthesis of program transformations", *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[34] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair", in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2019, pp. 8–18.

[35] X. Gao and A. Roychoudhury, "Interactive patch generation and suggestion", in *Automated Program Repair Workshop*, 2020.

[36] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey", *IEEE Trans. Software Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[37] "Github dependency graph", https://docs.github.com/en/code-security/supply-chain-security/about-the-dependency-graph, Accessed: 2021-04-09, 2021.

[38] S. Gulwani, "Automating string processing in spreadsheets using input-output examples", in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*, ACM New York, NY, USA, 2011.

[39] D. C. Halbert, "Programming by example", Ph.D. dissertation, University of California, Berkeley, 1984.

[40] S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, "Automatic android deprecated-api usage update by learning from single updated example", in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 401–405.

[41] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution", in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 274–283.

[42] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis", in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1159–1174.

[43] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response", in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 618–635.

[44] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches", in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 539–554.

[45] I. Jager and D. Brumley, "Efficient directionless weakest preconditions", in *Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab*, 2010.

[46] JetBrains, "IntelliJ", At https://www.jetbrains.com/idea/, 2020.

[47] JetBrains, "ReSharper", At `https://www.jetbrains.com/resharper/`, 2020.

[48] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis", in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, ACM, 2010, pp. 215–224.

[49] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code", in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2018, pp. 298–309.

[50] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", in *29th International Conference on Software Engineering (ICSE'07)*, IEEE, 2007, pp. 96–105.

[51] G. Jin, W. Zhang, and D. Deng, "Automated concurrency-bug fixing", in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX Association, 2012, pp. 221–236.

[52] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states", in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, 2014, pp. 315–326.

[53] R. Just, D. Jalali, and M. 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 (ISSTA)*, ACM, 2014, pp. 437–440.

[54] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (T)", in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2015, pp. 295–306.

[55] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing", *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[56] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[57] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation", in *Proceedings of the international symposium on Code generation and optimization (CGO)*, IEEE Computer Society, 2004, p. 75.

[58] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs", in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 216–226.

[59] V. Le, D. Perelman, O. Polozov, M. Raza, A. Udupa, and S. Gulwani, "Interactive program synthesis", *arXiv preprint arXiv:1703.03539*, 2017.

[60] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair", *Empirical Software Engineering (ESE)*, vol. 23, no. 5, pp. 3007–3033, 2018.

[61] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: Semantics-based repair of java programs via symbolic pathfinder", in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2017, pp. 376–379.

[62] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair", in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE Computer Society, 2016, pp. 213–224.

[63] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs", *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[64] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair", *IEEE Transactions on Software Engineering*, p. 54, 2012.

[65] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair", *Communications of the ACM*, vol. 62, no. 12, 2019.

[66] J. Lee, S. Hong, and H. Oh, "Memfix: Static analysis-based repair of memory deallocation errors for c", in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2018, pp. 95–106.

[67] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[68] F. Long and M. Rinard, "Staged program repair with condition synthesis", in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015, pp. 166–178.

[69] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems", in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 702–713.

[70] F. Long and M. Rinard, "Automatic patch generation by learning correct code", in *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[71] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors", in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014, pp. 439–452.

[72] Z. Manna and R. J. Waldinger, "A deductive approach to program synthesis", *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 90–121, 1980.

[73] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches", in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ACM, 2013, pp. 235–245.

[74] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, "User interaction models for disambiguation in programming by example", in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015, pp. 291–301.

[75] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation", *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 4, p. 15, 2018.

[76] S. Mechtaev, A. Griggio, A. Cimatti, and A. Roychoudhury, "Symbolic execution with existential second-order constraints", in *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ACM, 2018.

[77] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation", in *International Conference on Software Engineering (ICSE)*, 2018, pp. 129–139.

[78] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs", in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, IEEE, 2015, pp. 448–458.

[79] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis", in *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016, pp. 691–701.

[80] H. Mei and L. Zhang, "Can big data bring a breakthrough for software automation?" *Science China Information Sciences*, vol. 61, no. 5, p. 056 101, 2018.

[81] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example", in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, ACM, 2011, pp. 329–342.

[82] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples", in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE, IEEE Press, 2013, pp. 502–511.

[83] T. Mens and T. Tourwe, "A survey of software refactoring", *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb. 2004, ISSN: 0098-5589.

[84] Microsoft, "Visual Studio", At `https://www.visualstudio.com`, 2019.

[85] Microsoft, "Intellicode suggestions", At `https://docs.microsoft.com/en-us/visualstudio/intellicode/intellicode-suggestions`, 2020, (visited on 09/13/2020).

[86] "Micrsoft msbuild", `https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-api`, Accessed: 2021-04-15, 2021.

[87] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, "On the fly synthesis of edit suggestions", *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[88] T. M. Mitchell, "Generalization as search", *Artificial intelligence*, vol. 18, no. 2, pp. 203–226, 1982.

[89] M. Monperrus, "Automatic software repair: A bibliography", *ACM Comput. Surv.*, vol. 51, no. 1, 17:1–17:24, 2018.

[90] L. M. de Moura and N. Bjorner, "Z3: an efficient SMT solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[91] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution", in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 180–190.

[92] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation", *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[93] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis", in *Proceedings of the 35th International Conference onSoftware Engineering*, IEEE, 2013, pp. 772–781.

[94] W. F. Opdyke, "Refactoring object-oriented frameworks", UMI Order No. GAX93-05645, Ph.D. dissertation, Champaign, IL, USA, 1992.

[95] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, *et al.*, "Automatically patching errors in deployed software", in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 87–102.

[96] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution", in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[97] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing", in *2017 IEEE Symposium on Security and Privacy*, IEEE, 2017, pp. 615–632.

[98] G. D. Plotkin, "A note on inductive generalization", *Machine intelligence*, vol. 5, no. 1, pp. 153–163, 1970.

[99] A. Pnueli and R. Rosner, "On the synthesis of a reactive module", in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 179–190.

[100] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis", in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 107–126.

[101] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs", in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ACM, 2010, pp. 397–406.

[102] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization", in *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013, pp. 180–189.

[103] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair", in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[104] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems", in *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[105] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing", in *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[106] "Refazer: Program synthesis tool", https://www.nuget.org/packages/Microsoft.ProgramSynthesis, Accessed: 2020-09-21, 2020.

[107] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem", in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Springer, 1997, pp. 432–449.

[108] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in smt", in *International Conference on Computer Aided Verification*, Springer, 2015, pp. 198–216.

[109] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples", in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE, Buenos Aires, Argentina: IEEE Press, 2017, pp. 404–415.

[110] R. Rolim, G. Soares, R. Gheyi, T. Barik, and L. D'Antoni, "Learning quick fixes from code repositories", 2018. arXiv: 1803.03806 [cs.SE].

[111] "Roslyn framework", https://docs.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview, Accessed: 2021-04-15, 2021.

[112] A. Roychoudhury and Y. Xiong, "Automated program repair: A step towards software automation", *Science China Information Sciences*, vol. 62, no. 10, p. 200 103, 2019.

[113] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object oriented program repair", in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Computer Society, 2017, pp. 648–659.

[114] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software", in *23rd IEEE/ACM International Conference on Automated Software Engineering*, IEEE, 2008, pp. 218–227.

[115] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker", in {*USENIX*} *Annual Technical Conference (ATC)*, 2012, pp. 309–318.

[116] D. Shriver, S. Elbaum, and K. T. Stolee, "At the end of synthesis: Narrowing program candidates", in *IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track*, IEEE, 2017, pp. 19–22.

[117] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 43–54.

[118] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard, "Automatic discovery and patching of buffer and integer overflow errors",, 2015.

[119] R. Singh, "Blinkfill: Semi-supervised programming by example for syntactic string transformations", *Proc. VLDB Endowment*, vol. 9, no. 10, pp. 816–827, Jun. 2016.

[120] R. Singh and A. Solar-Lezama, "Synthesizing data structure manipulations from storyboards", in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 289–299.

[121] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair", in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, ACM, 2015, pp. 532–543.

[122] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. A. Saraswat, and S. A. Seshia, "Sketching stencils", in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation PLDI*, ACM, 2007, pp. 167–178.

[123] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures", in *Proceedings of the ACM SIGPLAN 2008 Conference on Pro-*

gramming Language Design and Implementation PLDI, ACM, 2008, pp. 136–148.

[124] A. Solar-Lezama, R. M. Rabbah, R. Bodik, and K. Ebcioglu, "Programming by sketching for bit-streaming programs", in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation PLDI*, ACM, 2005, pp. 281–294.

[125] A. Solar-Lezama, L. Tancau, R. Bodik, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs", in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, ACM, 2006, pp. 404–415.

[126] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps", in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 187–198.

[127] F. Tip, "A survey of program slicing techniques", *Journal of Programming Languages*, vol. 3, 3 1995.

[128] R. van Tonder and C. Le Goues, "Static automated program repair for heap properties", in *International Conference on Software Engineering (ICSE)*, ACM, 2018, pp. 151–162.

[129] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "Transit: Specifying protocols with concolic snippets", *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 287–296, 2013.

[130] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong", *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[131] M. T. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization", in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, ACM, 2010, pp. 327–338.

[132] Website, "American fuzzy lop (afl)", Accessed: 2019-04-08, 2019. [Online]. Available: `http://lcamtuf.coredump.cx/afl`.

[133] Website, “Bugzilla”, http://bugzilla.maptools.org/, Accessed: 2019-07-20, 2019.

[134] Website, “Cve”, https://cve.mitre.org/, Accessed: 2019-05-20, 2019.

[135] Website, “Oss-fuzz”, https://bugs.chromium.org/p/oss-fuzz, Accessed: 2019-05-22, 2019.

[136] Website, “Undefinedbehaviorsanitizer”, https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, Accessed: 2019-07-20, 2019.

[137] Website, “Libfuzzer - a library for coverage-guided fuzz testing”, https://llvm.org/docs/LibFuzzer.html, Accessed: 2018-12-21.

[138] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “Context-aware patch generation for better automated program repair”, in *International Conference on Software Engineering (ICSE)*, ACM, 2018, pp. 1–11.

[139] L. Xavier, A. Brito, A. Hora, and M. T. Valente, “Historical and impact analysis of api breaking changes: A large-scale study”, in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2017, pp. 138–147.

[140] Q. Xin and S. P. Reiss, “Identifying test-suite-overfitted patches through test case generation”, in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2017, pp. 226–236.

[141] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying patch correctness in test-based program repair”, in *Proceedings of the 40th International Conference on Software Engineering*, ACM, 2018, pp. 789–799.

[142] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, “Precise condition synthesis for program repair”, in *International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 416–426.

[143] S. Xu, Z. Dong, and N. Meng, “Meditor: Inference and application of api migration edits”, in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, IEEE, 2019, pp. 335–346.

[144] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs", *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[145] N. Yaghmazadeh, X. Wang, and I. Dillig, "Automated migration of hierarchical data to relational tables using programming-by-example", *Proc. VLDB Endow.*, vol. 11, no. 5, pp. 580–593, 2018.

[146] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 831–841.

[147] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the nopol repair system", *Empirical Software Engineering*, pp. 1–35, 2018.

[148] X. Zhu and A. B. Goldberg, "Introduction to semi-supervised learning", *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 3, no. 1, pp. 1–130, 2009.

[149] X. J. Zhu, "Semi-supervised learning literature survey", University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2005.