Synergising Program Analysis and Machine Learning for Program Repair

Nikhil Parasaram

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

October 21, 2024

I, Nikhil Parasaram, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work. The following chapters are based on the publications listed below:

- **Chapter 3:** Parasaram, Nikhil, Earl T. Barr, and Sergey Mechtaev. "Trident: Controlling side effects in automated program repair." IEEE Transactions on Software Engineering 48.12 (2021).
- **Chapter 4:** Parasaram, Nikhil, Earl T. Barr, and Sergey Mechtaev. "Rete: Learning namespace representation for program repair." 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023.
- **Chapter 5:** Parasaram, Nikhil, Earl T. Barr, Sergey Mechtaev, and Marcel Böhme. "Precise Data-Driven Approximation for Program Analysis via Fuzzing." In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 611-623. IEEE, 2023.
- **Chapter 6:** Parasaram, Nikhil, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl T. Barr, and Sergey Mechtaev. "The Fact Selection Problem in LLM-Based Program Repair." arXiv preprint arXiv:2404.05520 (2024). Submitted to a Conference, currently under review.

Abstract

Automated program repair (APR) enhances software quality by fixing bugs automatically, but it faces challenges due to software complexity. The vast number of possible patches makes exhaustive search impractical, and identifying correct patches is difficult since tools may generate incorrect fixes that overfit test cases. Mitigating these challenges involves leveraging the structure of code, which consists of a formal channel (execution semantics) and a natural language channel (comments, variable names). Machine learning excels at interpreting the natural language channel using large datasets but struggles with generating semantically correct patches. Conversely, program analysis provides detailed insights into program semantics. Combining program analysis with machine learning can address these challenges, using program analysis for execution specifics and machine learning for natural code aspects, like identifiers and comments. This thesis consists of four different works:

- Chapter 3 advances semantic repair by synthesizing patches with side effects, employing symbolic execution with state merging and effective patch prioritization to repair bugs in open-source projects.

- Chapter 4 reduces the search space by utilizing neural networks to learn variable information, ranking variables and patch templates to improve accuracy and reduce test overfitting. This enhances existing approaches, allowing them to repair previously unfixable bugs by leveraging program namespace information.

- Chapter 5 leverages abstract interpretation and fuzzing to probabilistically approximate reachable program states, focusing on high-probability states.

PSP boosts performance in abstract interpretation, symbolic execution, and patch prioritization, benefiting strategies discussed in Chapter 3 and Chapter 4.

- Chapter 6 addresses the challenge of identifying the most relevant facts, such as test errors and angelic values, for constructing effective prompts for LLM based APR. Extracted through program analysis, these facts build prompts whose effectiveness varies across different bugs. We develop a strategy to select facts tailored to each specific bug, significantly enhancing the effectiveness of LLMs in APR.

Impact Statement

With the advent of Generative AI, much research has focused on directly applying machine learning to various tasks, including automated program repair. While machine learning based techniques excel at generating intuitive patches, they often struggle to capture the relevant behavior during the execution of programs, especially in large real-world codebases. Program analysis, on the other hand, is crucial for bug fixing as it captures the semantic information of the program. This thesis leverages the combined strengths of program analysis and machine learning to enhance bug-fixing capabilities, creating a more robust and effective approach to automated program repair.

Some of the work in this thesis has been published in notable conferences and journals, including TSE, ICSE, and ASE. Notably, the research presented in Chapter 4, which leverages Machine Learning with program analysis to prioritise patches effectively, won the distinguished paper award at ICSE 2023.

Chapter 3 provides an efficient method of constructing a patch specification without restricting the patch type. This method helps speed up the process of validating a generated patch, making APR more efficient.

Chapter 4 provides a new representation of input structure for code. This representation can efficiently prioritise the variables to pick for a patch; this can help developers develop program repair tools or auto-complete tools for IDEs.

Chapter 5 improves program analysis, symbolic execution, and patch prioritisation in automated program repair by leveraging data obtained through fuzzing. This technique can be integrated into existing program analysis techniques, APR, and symbolic execution tools. The symbolic execution based optimisation was merged

into a symbolic execution engine named Mythril, which consists of 908k downloads and 3.8k stars to date.

The insights from Chapter 6, which explores the impact of different types of information (e.g., error messages, class and scope information, variable values) on bug fixing, can provide insights to developers who are interested in building an automated program repair tool that relies on Generative AI by helping them to understand which facts are important and help them select the most relevant facts for prompts.

Meta and Bloomberg have deployed Automated Program Repair systems, and companies such as Amazon, Uber, and Bytedance have been working on developing Generative AI-based agents to help improve developer productivity. This thesis will help provide insights to developers and researchers developing such APR tools.

Acknowledgements

I would like to express my deepest gratitude to my advisors, Sergey Mechtaev and Earl T. Barr, for their invaluable guidance and support throughout my research journey. Their proactive assistance and insightful advice have been quite important in my development as an effective researcher. I have learned numerous important skills under their mentorship, for which I am extremely thankful.

I am deeply grateful to Marcel Böhme for his guidance on research involving fuzzing, and for his assistance with chapter 5. His insights and support have been important for this work.

Additionally, I would like to extend my thanks to Valentin Wüstholz for his help with guiding me about instrumenting Harvey for evaluating my tool in chapter 5.

I am thankful to Yang Boyu and Yan Huijie for their invaluable help with constructing the framework that sends queries to OpenAI models for chapter 6 and extracting the data from the queries. Their efforts have significantly contributed to the completion of chapter 6.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Bugs are intrinsic to software development, presenting a challenging problem due to the inherent complexity of modern software. Large software projects often involve numerous files and thousands of lines of code, developed and maintained by multiple programmers over many years. This complexity is further amplified by constantly evolving software requirements, leading to frequent updates and the varying expertise levels of the programmers involved. As a result, identifying, diagnosing, and resolving bugs can be a tedious and error-prone process. Manual debugging demands substantial effort to understand the intricate interdependencies within the codebase and accurately locate and fix the root causes of errors. Therefore, Automated Program Repair (APR) tools can significantly improve both the efficiency and the quality of life for developers. This need has driven the development of various APR tools, such as Genprog [2], Prophet [3], and Angelix [4]. Moreover, some of these tools have seen successful deployment in the industry, including Getafix [5] at Meta and B-Assist [6] at Bloomberg.

A fundamental issue with Automated Program Repair (APR) is the vast number of candidate patches that must be considered, making the search space overwhelmingly large. Consider a simple program that checks if a number is within a certain range: `(x > 10 and x < 20)`. An APR tool might explore numerous modifications, such as changing > to >=, < to <=, or modifying `and` to `or`. Additionally, it could add complex conditions, like `(x > 10 and x < 20 or (y == 5))`, or introduce new logic. Each modification is a different candidate patch, and the com-

binations quickly become unmanageable. In real-world programs with thousands of lines of code, the number of candidate patches grows exponentially [1]. This immense search space makes the APR strategies relying on exhaustive search impractical, highlighting the need for effective search strategies to prioritise promising patches. Even if a patch passes the failing test case, which is used as the criteria for fixing the bug, it may end up overfitting that specific test without fixing the underlying problem. This issue is further elaborated in Section 2.1.4.

To effectively explore the search space, it is necessary to leverage the structure of the code. Code consists of two different channels [7]: a formal channel dealing with the execution semantics of the code and a natural language channel dealing with the understandability of the code (*i.e.* identifier names, comments, and docstrings). The formal channel can be interpreted through program analysis, which provides insights into the program's behavior and semantics. Conversely, the natural language channel can be interpreted by employing machine learning, which can interpret and learn from large datasets to recognise common coding patterns, conventions and idioms and exploit them effectively. By combining program analysis and machine learning, we can harness the strengths of both approaches. Program analysis will help with the generated patches satisfying the required execution semantics, while machine learning improves the naturalness and readability of the patches while reducing the likelihood of test overfitting. This dual-channel approach helps reduce the search space and increases the likelihood of generating correct and maintainable patches.

Several existing tools have leveraged program analysis for APR, such as Semfix [8], Angelix [4], and SPR [9]. Program analysis provides a robust framework for understanding code behavior, ensuring that generated patches maintain the intended functionality. Nonetheless, pure program analysis based techniques have notable limitations. One significant issue is the potential for overfitting to the success criteria, such as the failing tests, due to the large search space of candidate patches. Distinguishing between two non-equivalent patches that pass failing tests presents

[1] Super exponential *w.r.t.* the size of the namespace

a challenge. Some existing works addressed this issue by generating additional tests [10, 11]. However, using these techniques has drawbacks, such as the potential requirement for a large number of tests and performance issues when running a substantial amount of tests.

In contrast, Machine Learning (ML) has emerged as a powerful tool in APR [3, 12, 13], capturing the "naturalness" of code [14] by extensively training on existing open source repositories written by humans. This approach makes ML-generated patches more intuitive and easier for developers to understand. However, ML-based APR faces significant challenges. While ML models trained on this task excel at learning syntactic patterns, they often struggle to ensure whether the fixes they generate are correct, potentially leading to superficially correct patches that fail to address the underlying issue effectively. Additionally, most ML models typically operate within a fixed "window" of code due to computational and training complexity limitations. Transformer-based models, for example, require a manageable amount of input data to function effectively. They may struggle with processing larger amounts of code due to the quadratic relationship between the number of input tokens and the model's parameters. These window width limitations constrain the model's ability to fully understand the broader context of the code, potentially leading to incomplete or suboptimal repairs.

To address the challenges discussed above, this thesis presents these key contributions:

- **An Efficient Specification** Patches that have been generated by APR tools need to be validated. Most APR tools continue the process of generating and validating the generated patches until a patch that satisfies the criteria of success is found. Using test suite as the criteria of success is inefficient, as it takes a long time to rebuild the code with the generated patch and run the tests. Hence, prior works [4, 8, 15] used concolic execution to construct a patch specification. These tools validate the generated patches against the patch specification. These approaches cannot construct a patch specification for patches that cause side effects in the code. Chapter 3 provides an efficient

method of constructing the patch specification while alleviating this limitation.

- **Variable Prioritisation** Number of in-scope variables at any point in a codebase is massive. Hence, when constructing a patch, it is useful to have a method of prioritising variables. Identifying the best variables to fill a spot can be accomplished by using Machine Learning. However, ML based techniques suffer from window width limitation. Hence, this work proposes a new data structure named CDU chain to compress the snippet information to fit the window width. This variable prioritisation strategy can easily be plugged into existing APR techniques such as Prophet [3] and Trident, which is discussed in Chapter 3.

- **Patch Prioritisation using Static and Dynamic Information** Leverages fuzzing and abstract interpretation to construct a new data structure called Program State Probability (PSP). PSP, discussed in Chapter 5, can be used to prioritise patches effectively. Since PSP approximates the allowable program states, it has additional exciting applications, such as speeding up symbolic execution and serving as a knob, giving the ability to move across the space of over- and under-approximation of program behavior.

- **Study on Effect of Bug-Related Information**: This study, detailed in Chapter 6, is the first large-scale investigation of prompt design for Automated Program Repair (APR) using large language models (LLMs). Evaluating over 19,000 prompts with combinations of seven types of bug-related facts (e.g., GitHub issues, test errors, angelic values), the study reveals that adding more facts can degrade LLMs' bug-fixing performance, though each fact is shown to be useful under certain conditions. It also shows that dynamically choosing a subset of facts for different bugs is more effective than static selection. The study introduces a statistical model for optimising fact selection which outperforms static fact selection.

Research Scope

For the purposes of this research, a bug is defined strictly by the presence of a "failing test," ensuring a quantifiable criterion for what constitutes an error. Bugs that do not arise through failing tests, such as those identified through static analysis, code reviews, or runtime anomalies without explicit test failures, are not considered within this scope. Furthermore, the investigation assumes that the location of the bug, such as the specific function or line numbers, is already known; hence, techniques for bug localization, including methods for automatically identifying or predicting the buggy code regions, are outside the scope of this work. The scope is further confined to repairs involving single-function fixes. Initially, this is approached through repairs in a single chunk of code as discussed in Chapter 3, and subsequently, the scope expands to encompass function repairs in Chapter 6. This research does not address multi-function or cross-file fixes.

Chapter 2

Background

This chapter provides the essential background knowledge required to understand the methodologies and techniques employed in this thesis. It covers topics in three main areas: program analysis, machine learning, and automated program repair. The background related to automated program repair is discussed in Section 2.1, the program analysis techniques employed are discussed in Section 2.2, and the machine learning techniques are discussed in Section 2.3.

Chapter 3 mainly leverages symbolic execution discussed in Section 2.2.3 to make the patch specification extraction more efficient and additionally discusses a patch prioritisation algorithm to reduce test overfitting (Section 2.1.4). Chapter 4's primary contribution is that it leverages a new input representation with machine learning (Section 2.3) to improve variable selection. Chapter 5 uses Fuzzing (Section 2.2.1) and abstract interpretation (Section 2.2.2) to provide a new method of prioritising patches. Chapter 6 discusses the information to leverage when constructing a prompt to feed into LLMs (Section 2.3.2) for the program repair task.

2.1 Automated Program Repair

Automated Program Repair (APR) is the process of automatically generating a fix for a program such that it can be accepted by the developer. The goal of APR is to identify and correct defects in software without human intervention, thereby enhancing software reliability and reducing maintenance costs. The research community has dedicated increasing attention to this task [16].

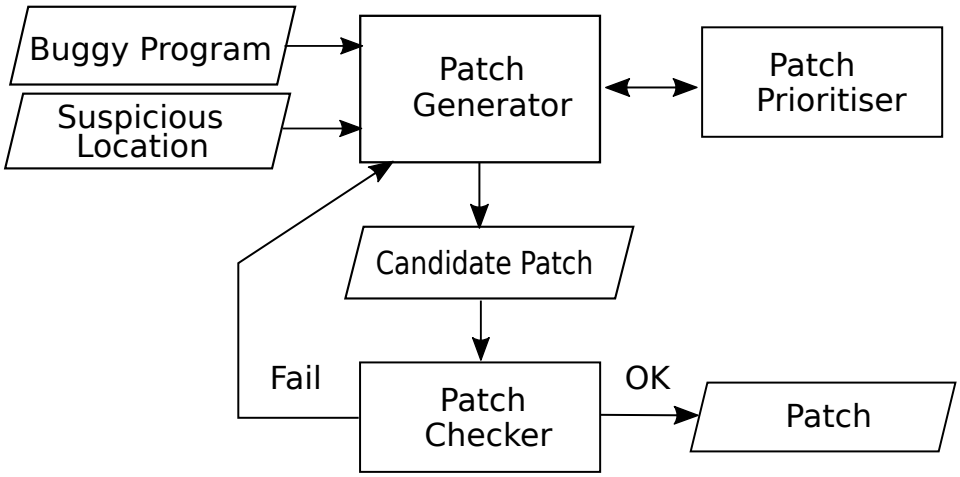


Figure 2.1: Architecture of standard APR tools. These tools take a buggy program and a suspicious location identified within the code. They generate a candidate patch and run it through a patch checker. If the candidate patch passes the checks, it is returned as the final patch. If it fails, the process iterates by generating and testing the next candidate patch until a successful patch is found or any termination criteria is reached.

2.1.1 Architecture

The architecture of standard Automated Program Repair (APR) tools consists of three primary components: the Patch Generator, the Patch Prioritiser, and the Patch Checker, as illustrated in Figure 2.1.

Each component plays a crucial role in the APR process. The Patch Generator is responsible for creating candidate patches. GenProg, a well-known APR tool, utilises genetic algorithms to generate patches [2]. Getafix, an APR tool developed by Meta, achieves this by mining fix patterns from multiple repositories and applying suitable patterns to the buggy locations [5]. Given the potentially large number of patches generated by these techniques, the Patch Prioritiser helps by ordering the candidate patches so that the most promising ones are ranked higher. This increases the likelihood of finding the correct patch efficiently. The candidate patches are then evaluated by the Patch Checker, which validates them against specific criteria, such as a test suite or a formal specification, to ensure their correctness.

2.1.2 Fault Localisation

Real-world repositories often contain tens of thousands of lines of code, making it impractical to generate patches without first identifying a set of potentially buggy lo-

cations. This is where fault localisation techniques come into play. Fault localisation techniques aim to identify the parts of the code that are most likely to contain bugs, thereby narrowing down the search space for the Patch Generator.

Fault localisation typically uses statistical methods to analyse program execution and identify suspicious code locations. These methods can include:

- **Spectrum-Based Fault Localisation (SBFL):** This technique involves running the program with a set of test cases and collecting coverage information. It calculates suspiciousness scores for each program element (*e.g.* statements, branches) based on the correlation between the elements executed and the test outcomes (pass or fail). Common measures used in SBFL include Tarantula [17], Ochiai [18], and Jaccard [18].

- **Machine Learning-Based Approaches:** Recent advancements have seen the application of machine learning techniques to fault localisation. These methods train models on information such as historical bug data, dynamic execution information such as code coverage and data dependencies *w.r.t.* failing tests to predict the likelihood of code regions being buggy [19, 20].

2.1.3 Correctness Criteria

The patch checker employs a correctness criteria to determine whether a candidate fix is plausible. To truly consider a patch correct, it should be able to pass a code review and be accepted by the developer. Finding a correct patch is achieved by validating the fix against a specific criteria which can be automatically checked against and comprehensive enough to ensure that all correct patches satisfy them. Given the challenges in obtaining formal specifications for real-world software, practical approaches typically rely on test suites [8, 4, 2, 3, 12]. Hence, the patches that pass through the patch checker are considered as plausible by APR tools. Some papers [4, 12] consider a generated plausible patch as correct when the generated patch is equivalent to the developer's patch.

In addition to test suites, static analysis tools can be used to evaluate candidate patches. These tools analyse the code without executing it to detect potential issues

such as security vulnerabilities, coding standard violations, and potential runtime errors. SymlogRepair [21] uses datalog based analysis as a correctness criteria for a patch. Tools such as InferFix [22] and GetaFix [5] employ static analysis along with failing tests to check whether the generated patch can be considered plausible.

2.1.4 Test Overfitting

Test overfitting is a common issue in Automated Program Repair (APR) where the generated patch passes all the provided test cases but fails to generalise to other untested scenarios. This occurs because the repair process is overly focused on satisfying the given test suite, potentially neglecting to fix the root cause.

To illustrate test overfitting, consider the following simple buggy program written in Python:

```
def find_max(a, b):
    if a > b:
        return b
    else:
        return a
```

(a) The Buggy program incorrectly returns the smaller value instead of the larger value.

```
# Test cases
assert find_max(5, 5) == 5
assert find_max(5, 10) == 10
assert find_max(-1, -5) == -1
```

(b) Test cases used for validating the program.

```
def find_max(a, b):
    if a == 5:
        return b
    else:
        return a
```

(c) Generated fix passes initial tests but fails to generalise.

Figure 2.2: Illustration of test overfitting in APR. The buggy program shown in (a) incorrectly returns the smaller value instead of the larger value. The initial test cases shown in (b) validate the generated fix shown in (c), which passes these tests but is logically incorrect.

The key problem with test overfitting is that the patch is validated against a limited set of tests, which may not cover the full range of behaviors required by the program. Consequently, the program may still contain bugs that were not exposed by the initial test suite.

Researchers have proposed various approaches to tackle this problem. The first group of approaches uses a pre-defined database of transformations to increase the chance of generating correct patch [23, 24, 25]. The second group generates additional tests [10, 11]. The third group of techniques prioritise the patches that are more likely to be correct [26, 27, 28, 29, 3]. This led to the development of a patch

prioritisation component for APR tools, as illustrated in the architecture depicted in Figure 2.1.

2.2 Program Analysis

This section discusses the program analysis techniques used in the thesis such as Fuzzing, Abstract Interpretation and Symbolic Execution.

2.2.1 Fuzzing

Fuzzing, also known as fuzz testing, is a dynamic software testing technique aimed at discovering vulnerabilities and bugs by automatically generating inputs to a computer program. The primary goal of fuzzing is to find security vulnerabilities and bugs by observing how the program behaves under these conditions. Formally, fuzzing can be defined as follows:

Definition 1 (Fuzzing)**.** *Fuzzing is a software testing technique that generates a set of test inputs I from the input space T and feeds them to a program p. The program is then executed with each input $i \in I$. Fuzzing monitors the program's execution for abnormal behavior like crashes, memory leaks, or assertion failures.*

The effectiveness of fuzzing depends on the ability to generate diverse and comprehensive input sets that can thoroughly test the various execution paths of the program p.

2.2.2 Abstract Interpretation

Abstract interpretation is a theory of over-approximating the semantics of computer programs [30]. It provides a framework for constructing abstract models of the program's behavior by mapping its concrete operations to an abstract domain. The primary goal of abstract interpretation is to perform static analysis to discover properties about the program that hold for all possible executions.

The motivation behind abstract interpretation is to analyze programs in a way that is both sound and computationally feasible. By working in an abstract domain, we can reason about the program's behavior without executing it (unlike fuzzing), thus enabling the detection of potential errors, such as bugs or security vulnerabilities,

before the program runs. One such vulnerability that can be analysed by abstract interpretation is null pointer exception. For instance, consider a program segment where a variable x can either be assigned a reference to an object or be null. In the abstract domain $\{NonNull, Null\}$, abstract interpretation can track the flow of values assigned to x throughout the program. If an operation is attempted on x when it can be *Null*, abstract interpretation can identify this potential error path. This capability allows developers to preemptively address null pointer exceptions during code analysis without executing the code.

Abstract interpretation involves a trade-off between precision and computational feasibility. In scenarios where precise determination of certain program properties is computationally infeasible (Rice's theorem [31]) or less critical, deliberately introducing imprecision into the analysis can be advantageous. For instance, in handling null pointer exceptions, focusing on whether a variable could be *Null* rather than precisely pinpointing every null pointer occurrence can still effectively highlight potential issues while reducing computational overhead. Similarly, for variables where only the sign matters, such as tracking whether a value is positive, negative, or zero, an abstract domain representing signs $(+, 0, -)$ simplifies analysis by abstracting away irrelevant details.

For a given concrete domain $(C, \leq_C)$ representing the possible states of the system and a set of concrete operations O describing the transitions between these states. Abstract interpretation aims to construct an abstract domain $(A, \leq_A)$ and establish a sound mapping across these domains.

Definition 2 (Abstract Interpretation)**.** *Abstract interpretation defines an abstract domain* $(A, \leq_A)$ *and operations* $O^\sharp$ *based on a concrete domain* $(C, \leq_C)$ *and their corresponding operations* O*. The abstraction function* $\alpha : 2^C \to A$ *maps concrete elements to an abstract element, and the concretisation function* $\gamma : A \to 2^C$ *maps abstract elements back to concrete elements. These functions satisfy the Galois connection property [30]:*

$$\forall a \in A, \alpha(\gamma(a)) = a \quad \textit{and} \quad \forall C' \subseteq C, C' \subseteq \gamma(\alpha(C'))$$

2.2.3 Symbolic execution

Symbolic execution is a program analysis technique in which a program is run with symbolic instead of concrete inputs. The result of this execution is a set of constraints over these symbolic variables which is called path constraint. SMT solvers are used to solve these path constraints to determine the feasibility of these paths. One of the main challenges of symbolic execution is the path explosion problem. For instance, with binary symbolic branching, the number of states doubles at each decision point, leading to a potential explosion in the number of feasible paths. To provide a precise understanding of symbolic execution, we first formally elaborate on SMT solving, which is required for resolving path constraints, and then formally define the semantics of symbolic execution

SAT/SMT solving

Boolean satisfiability problem is a problem of determining whether a propositional formula is satisfiable. Although this problem is NP complete, efficient algorithms were proposed like CDCL [32] which can handle large complex propositional formulas. Satisfiability Modulo Theories (SMT) is a decision problem of logical formulas with respect to a set of theories expressed in first-order logic. Examples of various theories include the theory of integers, the theory of arrays, the theory of bitvectors *etc.*

As is usual in SMT literature [33], we consider formulas and terms built from predicate and function symbols (e.g. "$+$", "$-$", "$>$") from a given signature Σ. We denote the set of all such formulas and terms as L_Σ. We also consider a background theory $\mathscr{T}$ that fixes the interpretations of the symbols in Σ.

We use the letters α, β, γ and δ to denote variables from L_Σ, and the letters π, ϕ and ψ to designate formulas from L_Σ. *Symbolic memory* θ is a function from memory addresses to logical terms from L_Σ (for an address a, the corresponding logical term is $\theta(a)$). We express the equality of two symbolic program states θ_1 and θ_2 for all initialised addresses as the formula $\theta_1 = \theta_2 \coloneqq \bigwedge_{a \in Initialised} \theta_1(a) = \theta_2(a)$.

Definition 3 (Satisfiability)**.** *$\{\alpha_1 \mapsto n_1, \ldots, \alpha_k \mapsto n_k\}$ is an assignment of the variables in a programming language $\mathscr{L}$ (a mapping from the variables to values). We say*

$$
\begin{array}{rcl}
\langle \text{stmt} \rangle & ::= & \langle \text{lvalue} \rangle = \langle \text{rvalue} \rangle \mid \langle \text{call} \rangle \mid \\
& & \texttt{if (}\langle \text{rvalue} \rangle\texttt{) \{ }\langle \text{stmt} \rangle\texttt{ \} else \{ }\langle \text{stmt} \rangle\texttt{ \}} \mid \\
& & \texttt{while (}\langle \text{rvalue} \rangle\texttt{) \{ }\langle \text{stmt} \rangle\texttt{ \}} \mid \\
& & \langle \text{stmt} \rangle\texttt{ ; }\langle \text{stmt} \rangle \\
& & \texttt{int }\langle \text{var} \rangle = \langle \text{rvalue} \rangle \\
& & \texttt{return }\langle \text{rvalue} \rangle \\
\langle \text{rvalue} \rangle & ::= & \langle \text{const} \rangle \mid \langle \text{var} \rangle \mid \star\langle \text{var} \rangle \mid \&\langle \text{var} \rangle \mid \\
& & \langle \text{rvalue} \rangle + \langle \text{rvalue} \rangle \mid \textit{other operations...} \\
\langle \text{lvalue} \rangle & ::= & \langle \text{var} \rangle \mid \star\langle \text{var} \rangle \\
\langle \text{rvlist} \rangle & ::= & \langle \text{rvalue} \rangle \mid \langle \text{rvalue} \rangle\texttt{ , }\langle \text{rvlist} \rangle \\
\langle \text{call} \rangle & ::= & \langle \text{fun} \rangle\texttt{ (}\langle \text{rvlist} \rangle\texttt{)} \\
\langle \text{vlist} \rangle & ::= & \langle \text{var} \rangle \mid \langle \text{var} \rangle\texttt{ , }\langle \text{vlist} \rangle \\
\langle \text{decl} \rangle & ::= & \texttt{int }\langle \text{fun} \rangle\texttt{ (}\langle \text{vlist} \rangle\texttt{) \{ }\langle \text{stmt} \rangle\texttt{ \}}
\end{array}
$$

Figure 2.3: Syntax of programming language $\mathscr{L}$.

that this assignment satisfies *a formula π iff a substitution of the variables α_i with the corresponding values n_i (denoted as $\pi[\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k]$) evaluates to True.*

Since it is inefficient, in general, to encode problems as SAT formulas, Satisfiability Modulo Theories addresses this by deciding the satisfiability of first order formulas with their given background theories. Each SMT formula can be transformed to an equivalent conjunctive normal form (CNF) using Tseytin algorithm [34]. Simply converting an SMT formula into CNF is not enough to apply a propositional satisfiability algorithm like CDCL, because SMT solvers need to evaluate not only the Boolean structure but also the theory-specific constraints. Specialized algorithms are required to combine Boolean reasoning with theory specific constraints corresponding to arithmetic, arrays, bitvectors, and other theories [33].

Formally Defining Symbolic Execution

A formal definition of symbolic execution is necessary to clearly explain the state merging strategy described in Section 3.3.2. For simplicity in formalism, we consider an imperative programming language $\mathscr{L}$ whose syntax is described in Figure 2.3. *Program* $p \in \mathscr{L}$ is a set of function declarations, i.e. $\mathscr{L} := 2^{decl}$, with distinct names.

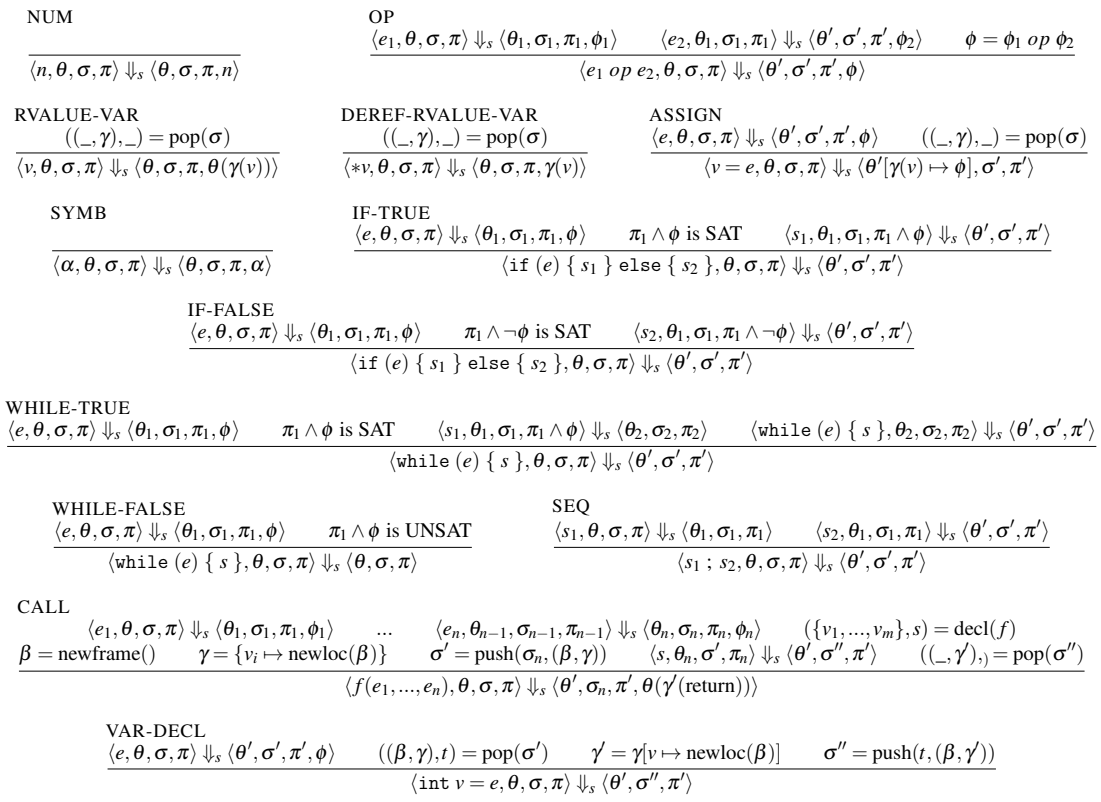


Figure 2.4: Semantics of symbolic execution.

$\mathscr{L}$ is a subset of the C programming language with only integer values, without global variables, and without switch statements, among other simplifications.

Memory is a function $\mu : \mathbb{N} \to \mathbb{Z}$ from addresses to values. *Stack frame* β is a bounded subset of $\mathbb{N}$. *Environment* γ is a mapping from variable names to their addresses. *Program stack* σ is a stack of pairs $\{(\beta_i,\gamma_i)\}_i$ of stack frame β_i and environment γ_i, with the usual stack operations *pop*, *pick*, and *push*. Stack frame allocator is a procedure *newframe* that, each time it is called, returns a new stack frame that is disjoint from previously allocated stack frames. Variable allocator *newloc* is a procedure that, for a given stack frame, returns a location within this stack frame that is not allocated to any variable.

Definition 4 (Semantics of $\mathscr{L}$[1])**.** *The semantics of a statement s in* $\mathscr{L}$ *is the relation* $\langle s,\mu,\sigma\rangle \Downarrow \langle\mu',\sigma'\rangle$*, where* μ' *and* σ' *are the memory and the stack obtained by executing the statement s in the context of memory* μ *and stack* σ *according to the*

[1]We omit, the rather standard, formal definition of the semantics of C-like language with pointers.

semantics of the C language. Similarly, the semantics of an rvalue expression e in $\mathscr{L}$ is the relation $\langle e, \mu, \sigma \rangle \Downarrow \langle \mu', \sigma', r \rangle$, where r is the result of evaluating the expression.

Programs in $\mathscr{L}$ are executed like C programs, under several simplifying assumptions: all values are allocated on stack, stack frames have infinite size and never de-allocated, and the entry function and its arguments have to be specified explicitly. Specifically, executing a program in $\mathscr{L}$ means evaluating the entry function applied to its arguments in the context of zeroed memory and an empty stack, as stated in the definition below:

Definition 5 (Execution)**.** *Let $p \in \mathscr{L}$ be a program, f be a function declared in p (entry function), $A \coloneqq [a_1, ..., a_n]$ be an ordered set of integers (entry function arguments). The function exec is defined as $exec(p, f, A) \coloneqq (\mu, r)$ such that $\mu_0 \coloneqq \lambda x.\ 0$, $\sigma_0 \coloneqq \emptyset$ and $\langle f(a_1, ..., a_n), \mu_0, \sigma_0 \rangle \Downarrow \langle \mu, \_, r \rangle$.*

The semantics of symbolic evaluation is defined as evaluation that transforms symbolic memory and augments current path constraint:

Definition 6 (Semantics of symbolic evaluation)**.** *The semantics of symbolic evaluation of a statement s in $\mathscr{L}$ is the relation $\langle s, \theta, \sigma, \pi \rangle \Downarrow_s \langle \theta', \sigma', \pi' \rangle$, where θ', σ' and π' are the symbolic memory, the stack and the path condition obtained by executing the statement s in the context of symbolic memory μ, stack σ, and path condition π according to the semantics of symbolic execution in Figure 2.4. The semantics of symbolic evaluation of rvalue expressions is defined accordingly (i.e. $\langle e, \theta, \sigma, \pi \rangle \Downarrow_s \langle \theta', \sigma', \pi', \psi \rangle$, where ψ is the symbolic result of evaluating the expression e).*

Definition 7 (Symbolic execution)**.** *Let $p \in \mathscr{L}$ be a program, f be a function declared in p (entry function), $\Phi \coloneqq [\phi_1, ..., \phi_n]$ be an ordered set of terms from L_Σ (entry function arguments). The function symexec is defined as $symexec(p, f, \Phi) \coloneqq \{(\pi_i, \theta_i, \psi_i)\}_i$ (set of triples) such that $\theta_0 \coloneqq \lambda x.\ 0$, $\sigma_0 \coloneqq \emptyset$ and for all i, $\langle f(\phi_1, ..., \phi_n), \theta_0, \sigma_0, True \rangle \Downarrow_s \langle \theta_i, \_, \pi_i, \psi_i \rangle$, where θ_i represents a symbolic mem-*

ory, π_i represents a path condition and ψ_i represents the symbolic result obtained by evaluating the function f.

2.3 Machine Learning

This section delves into two machine learning approaches used in this thesis: Random Forests and Transformers. Specifically, Section 2.3.1 explains random forests. Section 2.3.2 introduces language models which are used for predicting the most likely token to be the next. Finally, Section 2.3.3 delves into transformers, explaining their architecture.

2.3.1 Random Forests

Random forests are a machine learning method primarily used for classification and regression tasks. They operate by constructing a multitude of decision trees at training time and outputting the mode of the classes (classification) or mean prediction (regression) of the individual trees. Given that random forests comprise a multitude of decision trees, this subsection first delves into the decision trees in Section 2.3.1, explaining their construction and usage. Subsequently, the discussion progresses to random forests themselves in Section 2.3.1.

Decision Trees

A decision tree [35] is a flowchart-like structure where an internal node represents a feature (or attribute), the branch represents a decision rule, and each leaf node represents the outcome. The paths from the root to the leaf represent classification rules.

Mathematically, a decision tree is built by recursively splitting the data set D into subsets $D_1, D_2, \ldots, D_k$ with the goal of increasing homogeneity within these subsets. This means that we want each subset to be as pure as possible with respect to the target variable, either a class label in classification tasks or a continuous value in regression tasks.

Classification trees involve selecting splits that make the resulting child nodes contain instances that predominantly belong to a single class. The homogeneity is measured using criteria like Gini impurity or entropy [36]:

- **Gini impurity** quantifies the likelihood of an incorrect classification of a randomly chosen element if it was randomly labeled according to the distribution of labels in the node. A lower Gini impurity indicates a purer node.

- **Entropy**, derived from information theory, measures the amount of disorder or unpredictability in the node. Lower entropy indicates a higher level of certainty or purity within the node.

For each possible split, we calculate the impurity (using Gini or entropy) of the resulting child nodes and choose the split that results in the greatest reduction in impurity, thereby creating more homogeneous subsets.

Figure 2.5 illustrates a decision tree constructed on 150 samples of iris flowers[2] . In this example, the decision tree is tasked with classifying the iris flower species based on features such as sepal length, sepal width, petal length, and petal width. Decision trees partition the dataset by selecting features and thresholds that minimise the Gini impurity, a measure of class impurity or disorder, at each split. This process iteratively divides the samples into increasingly homogeneous subsets, ultimately leading to leaf nodes containing samples predominantly belonging to a single class.

For regression trees, the goal is to create subsets where the target values are as close to each other as possible. This is typically measured by the variance within the nodes:

- **Variance** measures how much the target values within a node deviate from the mean target value of that node. Lower variance means that the target values are closer to each other, indicating a more homogeneous node.

The best split is the one that minimises the variance in the child nodes, leading to groups of instances that have similar target values.

The decision tree algorithm aims to partition the data into subsets where the target variable (the variable we want to predict) is as uniform as possible within each subset. This process involves recursively selecting features with their corresponding

[2]This decision tree was constructed on the iris dataset [37], a widely used dataset in machine learning for illustrating various techniques. The iris dataset contains measurements of iris flowers, including sepal and petal dimensions, along with their corresponding species labels.

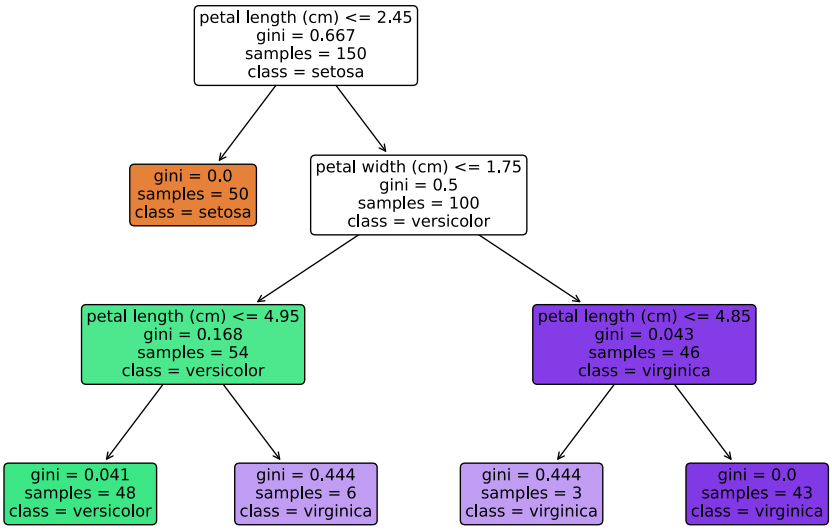


Figure 2.5: A decision tree depicting the classification of iris flower species. Decision trees partition the dataset by selecting features and thresholds that minimise the Gini impurity, a measure of class impurity or disorder, at each split. This process iteratively divides the samples into increasingly homogeneous subsets, ultimately leading to leaf nodes containing samples predominantly belonging to a single class.

thresholds that minimise the impurity or disorder of the target variable at each split. The goal is to create increasingly homogeneous subsets where the majority of samples belong to the same class or category. This recursive splitting continues until a stopping criterion is met, such as reaching a maximum tree depth or having a minimum number of samples per leaf node.

Random Forests

Random forests improve the accuracy and robustness of a single decision tree by combining multiple trees. Each tree in the forest is built by sampling with replacement from the training data. This technique of sampling with replacement on training data is called bootstrap sampling. Additionally, when splitting a node, a random subset of features is considered, which introduces diversity among the trees.

Let $\{T_1, T_2, \ldots, T_B\}$ be the B decision trees in the forest, each trained on a bootstrap sample of the data D. For classification, the forest predicts the class label $\hat{y}$ by majority voting:

$$\hat{y} = \arg \max_{c \in \{1,\ldots,C\}} \sum_{b=1}^{B} I(T_b(x) = c)$$

where $I(\cdot)$ is the indicator function that is 1 if the argument is true and 0 otherwise, and $T_b(x)$ is the class prediction of the b-th tree for input x.

For regression, the prediction $\hat{y}$ is given by averaging the predictions of the individual trees:

$$\hat{y} = \frac{1}{B} \sum_{b=1}^{B} T_b(x)$$

2.3.2 Language Models

Definition 8 (Language Model)**.** *A* language model *is a probabilistic model that assigns a probability $P(w_1, w_2, \ldots, w_n)$ to a sequence of words $(w_1, w_2, \ldots, w_n)$ in a language. Formally, a language model estimates the likelihood of a word sequence by factorizing it using the chain rule of probability:*

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

where $P(w_i \mid w_1, w_2, \ldots, w_{i-1})$ is the conditional probability of the word w_i given the preceding words $(w_1, w_2, \ldots, w_{i-1})$.

In practice, language models often employ approximations, such as the Markov assumption, simplifying the model by considering only the previous state [38]. Specifically, bigram models, a type of n-gram language model where $n = 2$, consider only the previous word. More generally, n-gram language models consider the probability of the current word w_i given the previous $n-1$ words:

$$P(w_i \mid w_1, w_2, \ldots, w_{i-1}) \approx P(w_i \mid w_{i-n+1}, \ldots, w_{i-1}).$$

Earlier language models, such as n-grams, relied on statistical estimation techniques

such as Maximum Likelihood Estimation (MLE) to estimate these probabilities [38]. However, as machine learning techniques advanced, language models evolved to encompass more sophisticated approaches, such as leveraging neural networks (*e.g.* RNN, LSTM, Transformers).

Although language models were initially developed for natural languages, they have also been applied to formal languages, such as programming languages [14, 39], as well as to symbolic domains like mathematical theorem proving, where structured patterns and rules govern the reasoning process [40, 41].

In the context of machine learning, parameters refer to the variables that the model learns from data during the training process. These parameters are adjusted to minimise the difference between the model's predictions and the actual data. Language models with a large number of parameters (*e.g.* 300 Million) are known as large language models (LLMs). LLMs excel at capturing intricate patterns and nuances in language due to their extensive parameterisation, which is typically achieved through the use of large neural networks. They are fundamental to many natural language processing tasks, including speech recognition, machine translation, and text and code generation.

2.3.3 Transformers

Transformers are a type of neural network architecture introduced in the paper "Attention is All You Need" [1]. Transformers have revolutionised the field of natural language processing (NLP) by providing the foundation for many state-of-the-art models, including BERT and GPT. Transformers are particularly effective in building language models as they can handle long-range bi-directional dependencies between the tokens and can be parallely trained. Large language models (LLMs) are typically constructed using the transformer architecture, enabling them to achieve remarkable performance across various NLP tasks, such as text and code generation, speech recognition, and machine translation.

Architecture Overview

Transformers are designed to handle sequential data, but unlike traditional recurrent neural networks (RNNs) [42] and long short-term memory networks (LSTMs) [43], they do not process the data in order. Instead, transformers rely entirely on a mechanism called *self-attention* to draw global dependencies between inputs. The architecture of transformers, proposed by Vaswani et al [1], comprises multiple layers, with each layer incorporating either attention or a feedforward layer. This design, as depicted in Figure 2.6, enables the model to capture intricate relationships between different parts of the input sequence while also allowing for non-linear transformations through the feedforward neural networks.

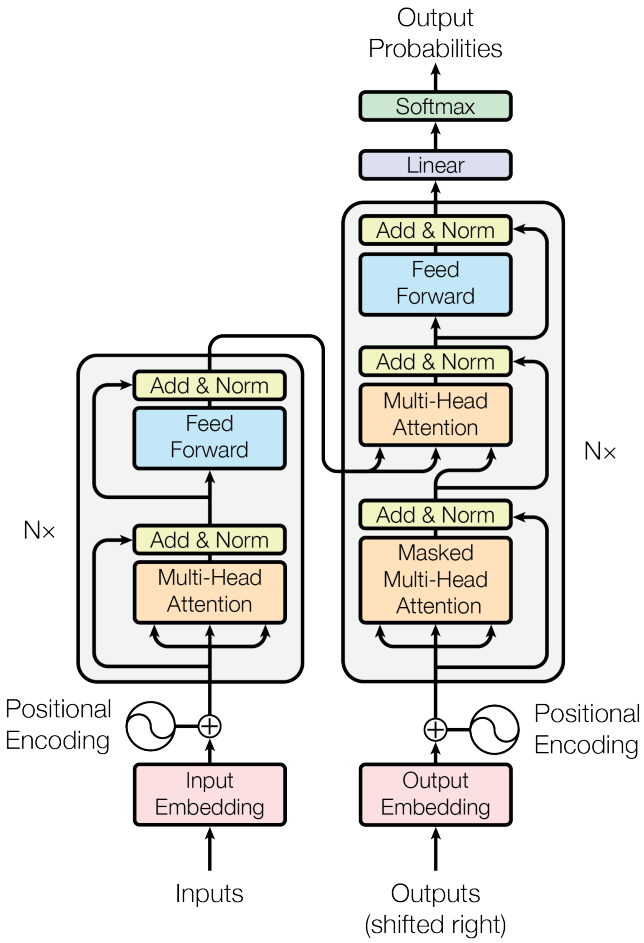


Figure 2.6: This figure taken from Vaswani et al [1] illustrates the architecture of transformers.

Self-Attention Mechanism

The self-attention mechanism allows each position in the input sequence to attend to all other positions, computing a weighted sum of these positions. The weights are determined by a similarity function between the positions.

For an input sequence of length n, each element is usually represented by a vector, known as an embedding. While embeddings are typically dense vectors—meaning most positions contain non-zero values—they can also be sparse, depending on how they are constructed. Dense embeddings are often preferred because they capture richer semantic relationships by placing similar elements closer together in the continuous vector space. This allows the encoding of various linguistic or contextual features of the token, such as syntactic or semantic meaning, which is more challenging to achieve with sparse or one-hot encodings. These embeddings are denoted as $x_1, x_2, \ldots, x_n$. The self-attention process involves three steps:

- **Query, Key, and Value Vectors**: Each input embedding x_i is projected into three vectors: a query q_i, a key k_i, and a value v_i. These vectors are computed using learned weight matrices W^Q, W^K, and W^V:

$$q_i = W^Q x_i, \quad k_i = W^K x_i, \quad v_i = W^V x_i$$

  - **Query vector** q_i Represents the current position's vector used to query the other positions.
  - **Key vector** k_i: Represents the vector against which the query is compared to compute attention scores.
  - **Value vector** v_i: Represents the information to be aggregated based on the attention scores.

- **Attention Scores**: The attention score for each pair of positions (i, j) is computed as the dot product of the query from position i and the key from position j and scaled by the square root of the dimension of the key vector, denoted as d_k:

$$Attention(q_i, k_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

$q_i \cdot k_j$ represents the dot product between the query vector q_i and the key vector k_j. This dot product measures the alignment of the query and the key. The higher the dot product, the more relevant the key is to the query. The scaling by d_k is used to mitigate the effect of large dot products due to high dimensionality.

- **Attention Weights and Output**: Once the attention scores are computed, they are passed through a softmax function to obtain the attention weights. These attention weights represent the importance or relevance of each position in the input sequence with respect to the current position. The softmax function ensures that the attention weights sum up to 1, allowing for a proper weighting of the value vectors during the aggregation step. The attention weights are then used in computing a weighted sum of the value vectors, resulting in the output of the attention mechanism:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here, Q, K, and V are matrices containing the query, key, and value vectors for all positions in the sequence. To prevent information from future positions influencing the current position (maintaining the auto-regressive property), the dot products corresponding to the future to past information flow (*i.e.* right to left) are set to $-\infty$ before applying the softmax function. This masking ensures that the attention mechanism respects the temporal order of the sequence during training and inference.

Multi-Head Attention

Instead of using a single attention function, transformers employ multi-head attention. This means that the model utilises multiple sets of query, key, and value vectors. If a transformer model with eight attention heads, each head can independently focus on

different aspects of the input data, such as identifying relationships between words or capturing syntactic structures.

Multiple attention heads are handled through concatenation as follows:

$$MultiHead(Q,K,V) = Concat(head_1, head_2, \ldots, head_h)W^O$$

where each $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is a learned projection matrix. The projection matrices used in the attention mechanism are:

- W_i^Q: Query projection matrix for head i, used to transform the input into the query vector space.
- W_i^K: Key projection matrix for head i, used to transform the input into the key vector space.
- W_i^V: Value projection matrix for head i, used to transform the input into the value vector space.

Positional Encoding

Since transformers do not inherently capture the sequential order of the input data, positional encodings are added to the input embeddings to provide information about the position of each token in the sequence. These encodings can either be learned or predefined.

Feed-Forward Networks and Layer Normalization

Each layer in the transformer encoder includes a fully connected feed-forward network (FFN) applied to each position separately and identically. This is followed by layer normalization and residual connections to stabilize and improve training:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

where W_1, W_2, b_1, and b_2 are learned parameters.

Chapter 3

Trident: Controlling Side Effects in Automated Program Repair

3.1 Introduction

Most existing software is written in imperative programming languages. Statements in imperative languages can have side effects: observable effects beyond returning a value to the invoker of the operation. Common side effects include changing the value of a variable, writing data to disk, or enabling or disabling a button in the user interface.

Despite the importance of modifications with side effects for real-world programs, state-of-the-art program repair tools have a limited ability to handle side effects, which restricts their applicability. Test-driven program repair approaches that rely on program synthesis, both symbolic [15] and enumerative [9], currently only synthesise side-effect-free expressions, or scale only to small programs [44]. Current heuristic repair approaches generate patches with side effects only if suitable code fragments exist in the buggy program [45] or in a pre-defined database of patterns [24], which may not contain the needed code. Machine learning approaches [46] can potentially generate patches with side effects, but their success depends on the size and the quality of training data. Finally, existing approaches for controlling test-overfitting [26, 29] do not take side effects into account. This lacuna is important, because our experiments demonstrate that patches with side effects are

more prone to test-overfitting.

This chapter introduces TRIDENT, the first test-driven program repair approach that synthesizes patches with side effects without relying on the plastic surgery hypothesis, a database of patterns, or training data. TRIDENT effectively addresses the limitations of previous constraint-based program repair techniques [4, 15], and complements heuristic techniques [45, 23], since it does not rely on the plastic surgery hypothesis [47] or a repair pattern database.

TRIDENT takes a buggy program and its test suite, then follows the general workflow of semantic program repair [8]: it localises faulty statements, infers a specification for patching these statements using symbolic execution, and then constructs a patch via program synthesis. Assignment statements and function calls are two basic classes of statements that involve side effects. Compared to previous semantic techniques, TRIDENT supports two new defects classes: the insertion/modification of assignment statements and function calls. To repair a software defect, an automated program repair tool must (1) contain a correct patch in its search space, (2) find this patch within a time budget, and (3) reduce the chance of generating an incorrect, test-overfitting patch. TRIDENT tackles these challenges by enhancing specification inference and patch synthesis for assignment statements and function calls.

In order to include patches with side effects in its search space, TRIDENT extends existing component-based program synthesis approaches [48, 49] by introducing the concept of lvalue components, which can appear in the left-hand-sides of assignments, and rvalue components, which can appear in the right-hand-sides of assignments. Then, the basic building block of TRIDENT's patch synthesiser is a k-holed assignment that represents a simultaneous assignment of k rvalue components to k lvalue components (Section 3.3.2). This formalism captures the semantics of both assignment statements and function calls with side effects. Functions without loops can be precisely summarised as simultaneous assignments [50]. For functions with loops, TRIDENT computes summaries using loop unrolling.

The inclusion of patches with side effects in the search space causes a scalability

problem during the symbolic execution that infers the specification of the holes because it exacerbates the path explosion problem. Side effects introduce additional state changes that must be tracked during concolic execution. Each side effect can alter the program's state, resulting in new execution paths and contributing to path explosion. For example, if the task is to model a missing statement that updates an unknown variable with an unknown value at a particular location, performing symbolic execution from this point, with a symbolic assignment of an unknown variable at an unknown location in the memory to an unknown value, generates new program states for each potential variable/memory location that could have been modified, leading to a rapid increase in execution paths. To alleviate this problem, TRIDENT leverages a novel merging strategy that rests on two insights. First, distinct variables can be updated and still generate states that traverse the same path to program exit. Second, even when many variables can be assigned, any concrete patch will affect only a few of them. When two paths write the same thing (the same rvalue) to two different variables (two different lvalues), TRIDENT exploits the first insight to efficiently merge both writes into a single state. To ensure consistency of this merging, TRIDENT appends an appropriate path constraint to the path condition (Section 3.3.2). It exploits the second insight to restrict the number of variables that a patch can update.

Although many bugs in imperative programs require side-effected patches to fix, some side effects can increase overfitting as our experiments demonstrate (Section 3.5). To address this issue, TRIDENT applies a simple patch prioritisation heuristic to minimise overfitting. We find that preferring a patch with the fewest side effects lowers overfitting.

To evaluate TRIDENT, we used three benchmarks: 10 bugs extracted from free GNU projects for evaluating TRIDENT's scalability on bugs that require patches with side effects, 36 bugs sampled from ManyBugs [51] for evaluating TRIDENT's scalability on generic defects and 110 defects extracted from Codeflaws benchmark [52] for evaluating propensity of TRIDENT's patches to overfit. The 10 bugs were the first 10 sampled uniformly whose fixes required side effects and the 110 were cut

down from the complete Codeflaws dataset, again to those needing side effects (Section 3.5). The evaluation demonstrates that TRIDENT generates patches involving assignments and function calls for 6 out of the 10 realistic bugs. Furthermore, TRIDENT's patch prioritisation increases the rate of correct patches from 33.3% to 58.3% when applied to the 110 defects from Codeflaws. These results demonstrate the practicality and utility of TRIDENT.

The key contribution of this work is TRIDENT, the first scalable test-driven patch synthesis approach that addresses the memory updates/call function defect class without relying on the plastic surgery hypothesis, a database of patterns, or training data; it is enabled by a tight integration of two technical insights:

- An extension of component-based program synthesis that introduces lvalue and rvalue components to capture assignments and function calls;
- Multi-path specification inference, a state merging technique tailored to the synthesis of side effected patches that mitigates path explosion.

All code, scripts, and data necessary to reproduce this work are available at `https://github.com/norhh/Trident-TSE`.

3.2 Overview

For a given buggy program, test-driven program repair (TPR) techniques search for a plausible patch — a patch that passes a test-suite — in a search space of candidate patches. Since a test-suite is an incomplete specification, program repair systems utilize patch prioritization strategies to increase the probability of finding a correct patch. Thus, to repair a bug, a TPR system has to (1) contain the correct patch in its search space, (2) be efficient enough to find a plausible patch given a time budget, and (3) prioritize a correct patch over plausible, but incorrect, patches.

TRIDENT is the first TPR technique to synthesise side-effected patches. This fundamental advance exacerbates all three of TPR's seminal problems. Section 3.2.1 presents the resulting challenges; the rest of the section overviews how TRIDENT overcomes them. TRIDENT's novel primitive is its notion of k-holed assignment

```
int clamp(int x, int l,
    int h) {
    int r;
    // missing
        assignment r =
        x;
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

(a) A function missing an assignment.

```
int clamp(int x, int l,
    int h) {
    int r;
    □_1^L,...,□_k^L = □_1^R,...,□_k^R;
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

(b) A buggy function with holes, placed before suspicious statements.

```
int clamp(int x, int l,
    int h) {
    int r;
    switch(PATCH_ID) {
        case 0: x =
            □^R; break;
        case 1: l =
            □^R; break;
        case 2: h =
            □^R; break;
        case 3: r =
            □^R; break; }
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

(c) Eliminating lvalue-holes using a switch statement (for $k = 1$).

Figure 3.1: Synthesising a patch that inserts an assignment statement.

(Section 3.2.2), which can capture function calls (Section 3.2.3). Section 3.2.4 details TRIDENT's strategy for resisting overfitting.

3.2.1 The Challenges of Synthesis with Side Effects

The two key challenges of adding the insertion/modification of assignment statements into the TPR search space are (1) efficiency, i.e. how to efficiently search the extended space for plausible patches, and (2) test-overfitting, i.e. how to ensure that the generated patches are correct.

Consider the function `clamp` in Figure 3.1a; it restricts a number between two other numbers. It has a bug: it does not assign `r` to `x` before its checks. Suppose `clamp` fails the test $\texttt{clamp}(2,1,4) \rightarrow 2$, where the " $\rightarrow 2$" denotes the expected output.

The first way to automatically repair this bug is *generate-and-validate* program repair: enumerate and test all possible insertions of assignments of the form $v = e$, where $v \in \{\texttt{x}, \texttt{l}, \texttt{h}, \texttt{r}\}$, the program variables in scope, and e is an expression over these variables. The main shortcoming of this method is that it requires a large number of test executions, and therefore does not scale to the large search spaces needed to repair realistic bugs. The second way to automatically repair this bug is

```
int buggy(int x, int y)
    {
    // missing call
        inc_if_zero(&x,&y)
    if (x > 0 && y > 0)
        return 1;
    else
        return 0;
}
```

(a) A function missing a function call.

```
void inc_if_zero(int &x,
     int &y) {
    if (x == 0)
        x++;
    if (y == 0)
        y++;
}
```

(b) The definition of `inc_if_zero`.

```
int buggy(int x, int y)
    {
    // Hole precedes
        suspicious code
    □1L, □2L = □1R, □2R;
    if (x > 0 && y > 0)
        return 1;
    else
        return 0;
}
```

(c) Assignment synthesis in TRIDENT.

Figure 3.2: Synthesizing a patch that inserts a function call.

synthesis-based program repair. Existing techniques, such as Angelix [4], do not synthesize assignments, because they are limited to side-effect-free expressions.

Since a test suite is an incomplete specification, TPR techniques are prone to test-overfitting, generating patches that pass the tests, but are incorrect. Extending the search space with side-effected patches poses additional challenges: (1) larger search spaces are more prone to test-overfitting [53], and (2) intuitively, changes with side effects are more likely to break functionality that is not covered by tests. A common approach to alleviate test-overfitting is to define a cost function on the search space, and search for a patch that passes the tests and minimises the cost. To the best of our knowledge, no cost function that takes side effects into account has been proposed.

3.2.2 Synthesising Assignments

To address the efficiency challenges that side-effected patches pose, we first introduce a new reification of a memory update that we call *k*-holed assignment. Then, we use this representation to define an efficient update-aware specification inference approach called multi-path specification inference.

A *lvalue-hole*, or $\square^L$, refers to a set of writable memory locations. The two-holed assignment $\square^L = \square^R$ combines an lvalue-hole and an rvalue-hole. Synthesising such assignment effectively means filling $\square^L$ with an lvalue (e.g. a variable), and $\square^R$ with an rvalue (e.g. an arithmetic expression). A *k-holed assignment* $\square^L_1, ..., \square^L_k = \square^R_1, ..., \square^R_k$ generalises two-holed assignment to the simultaneous assignment of up to *k* lvalue

holes (Definition 13).

Following SemFix [8], TRIDENT inserts holes before suspicious locations (Figure 3.1b and Figure 3.2c), infers a logical specification for the holes, and then leverages the inferred specification to synthesise expressions to fill the holes. SemFix specification inference uses symbolic execution equipped to handle programs with rvalue-holes. Thus, a key prerequisite for supporting synthesis with side effects is extending specification inference to programs with k-holed assignments, as shown in Figure 3.1b.

A naïve approach to extend specification inference to k-holed assignments is to transform these assignments into switch statements as shown in Figure 3.1c for $k = 1$. Specifically, we can enumerate all possible variables that can appear on the left-hand side of an assignment as cases in a switch statement, and, for each case, insert an rvalue-hole for the right-hand expression. This transformation allows reusing SemFix' specification inference to synthesize an assignment statement. However, this approach is inefficient because adding a switch statement significantly increases the number of paths that symbolic execution must explore. Current general-purpose state-merging strategies do not efficiently handle the resulting path explosion because they require fixed state topology and do not have a mechanism for bounding the search of lvalues. Section 7.1 elaborates on these limitations.

To alleviate path explosion of this naïve approach, we propose a more efficient semantics for inferences the specification of a k-holed assignment that we call *multi-path specification inference*. Multi-path specification inference rests on the insight that assignments to different variables along a path can leave that path's decisions unchanged. For example, inserting `x = l-1;` or `l = x+1;` at entry in Figure 3.1a both result in executing the true branch of first if-statement and the false branch of the second if-statement. Therefore, it is possible to *merge* the states induced by assignments to different variables, significantly reducing the number of paths to explore, thereby increasing the chance of finding a patch within a time budget. After merging states that correspond to assignments of different variables, the resulting path constraint effectively captures an equivalence class of assignments

that drive test execution along this path.

To merge states corresponding to assignments of different variables, multi-path specification inference uses two groups of constraints: (1) those representing an assignment of a symbolic value to each writable memory location, controlled by a dedicated Boolean selector variable, and (2) cardinality constraints over the selector variables that restrict the number of memory locations that a patch satisfying the specification assigns along a given path.

Consider the k-holed assignment statement $\square_1^L, \ldots, \square_k^L = \square_1^R, \ldots, \square_k^R$ at the missing assignment in Figure 3.1b. Multi-path specification inference, when executing this statement, constructs the following path constraint:

$$\begin{aligned}\phi \stackrel{\text{def}}{=} & (s_x \rightarrow x' = \alpha_x \wedge \neg s_x \rightarrow x' = x) \\ & \wedge (s_l \rightarrow l' = \alpha_l \quad \wedge \neg s_l \rightarrow l' = l) \\ & \wedge (s_h \rightarrow h' = \alpha_h \wedge \neg s_h \rightarrow h' = h) \\ & \wedge (s_r \rightarrow r' = \alpha_r \ \wedge \neg s_r \rightarrow r' = r) \\ & \wedge \mathit{AtMost}(k, s_x, s_l, s_h, s_r)\end{aligned}$$

where, for $\beta \in \{x, l, h, r\}$, α_β is the symbolic variable representing values of the rvalue-hole $\square^R$, s_β is a Boolean selector variable that enables/disables assignments to different memory locations, β' is the value of the program variable β after the statement, and $\mathit{AtMost}(k, -)$ is a cardinality constraint that encodes that at most k of its program variable arguments can be *True*. Here, $\mathit{AtMost}(k, -)$ means that at most k of the variables `x`, `l`, `h` and `r` can be modified as a result of executing the k-holed assignment.

TRIDENT's encoding merges the semantics of a subspace of assignments into the same state. Then, the values of the selector variables determine the subset of variables the synthesised statement updates. This significantly reduces the number of explored paths during symbolic execution. For example, consider a path that takes the false branches of the both if-statements that corresponds to the constraint $\psi \stackrel{\text{def}}{=} (x' \geq l') \wedge (r' \neq x' \vee x' \leq h')$. When TRIDENT explores this path in the program

with k-holed assignments (Figure 3.1b), the path has the path condition $\phi \wedge \psi$. Lacking k-holed assignment, traditional symbolic execution would, to achieve an equivalent result, have to explore four paths in the program with switch statement (Figure 3.1c), yielding the path conditions $x' = \alpha_x \wedge \psi$, $l' = \alpha_l \wedge \psi$, $h' = \alpha_h \wedge \psi$ and $r' = \alpha_r \wedge \psi$. In this case, TRIDENT reduces the number of explored paths, for $k = 1$, from 4 to 1.

After exploring the path $\phi \wedge \psi$, TRIDENT can synthesize the patch `r = x`, since this patch is consistent with $\phi \wedge \psi$ if s_r is true, and produces the desired output $r' = 2$. To synthesize it, TRIDENT uses the variables {x, l, h, r} as both lvalue and rvalue components. TRIDENT automatically extracts the variables and data fields defined or used in the current function as components, as Section 3.3.3 details.

The cardinality constraint $AtMost(k, -)$ is essential for assignment synthesis. First, it allows synthesising statements that modify more than one memory location. Second, it reduces the search space by avoiding paths that are only feasible when more than k variables are modified. For example, consider `clamp` in Figure 3.1b and the test $\texttt{clamp}(2,1,4) \rightarrow 2$, which `clamp` fails returning 0, not 2. Using $AtMost(1, -)$ to restrict the search space to assignments that modify at most one variable makes the path that follows the false branch of the first if-statement and the true branch of the second if-statement infeasible. This is because changing evaluation of the second if-statements requires changing the binding of two variables, which $AtMost(1, -)$ forbids because it makes the following constraint unsatisfiable:

$$
\begin{aligned}
&(s_x \rightarrow x' = \alpha_x \wedge \neg s_x \rightarrow x' = 2) \\
\wedge\ &(s_l \rightarrow l' = \alpha_l \wedge \neg s_l \rightarrow l' = 1) \\
\wedge\ &(s_h \rightarrow h' = \alpha_h \wedge \neg s_h \rightarrow h' = 4) \\
\wedge\ &(s_r \rightarrow r' = \alpha_r \wedge \neg s_r \rightarrow r' = 0) \\
\wedge\ &AtMost(1, s_x, s_l, s_h, s_r) \\
\wedge\ &(x' \geq l') \\
\wedge\ &(r' = x' \wedge x' > h')
\end{aligned}
$$

We assume that k is relatively small, because large k implies that complex modifications are required and such programs have serious problems such as wrong algorithms or lacking functionality. TRIDENT is designed for program features that are almost correct with the exception of a small fragment of code. Currently, TRIDENT starts from $k = 1$. If it cannot synthesise a patch, it increments k and repeats, until it reaches a configurable bound on k.

3.2.3 Synthesising Function Calls

The abstraction provided by k-holed assignments is powerful enough to express more complex, side-effected modifications such as function calls. This is because loop-free functions are equivalent to simultaneous assignment, and program with loops can be summarised as simultaneous assignments using loop unrolling [50].

Consider the buggy program in Figure 3.2a with a missing call to the function `inc_if_zero` shown in Figure 3.2b. Assume `buggy` fails the test $\texttt{buggy}(0,0) \rightarrow 1$.

TRIDENT computes function summaries for all functions that can be called at the target location. Currently, user must provide a library of functions, and TRIDENT computes the summaries via symbolic execution with loop unrolling. For example, TRIDENT computes the following summary for `inc_if_zero`:

$$(x=0 \rightarrow x'=x+1 \wedge x\neq 0 \rightarrow x'=x)$$
$$(y=0 \rightarrow y'=y+1 \wedge y\neq 0 \rightarrow y'=y)$$

where x and y denote the values bound to the variables `x` and `y` before executing `inc_if_zero`, and x' and y' represent their values after executing it.

Given function summaries, TRIDENT executes `buggy` to infer its patch synthesis specification, as explained in Section 3.2.2. Here, we assume that TRIDENT infers this specification with $\mathit{AtMost}(k = 2, -)$, which corresponds to inferring a specification for lvalue-holes in the form of simultaneous assignment to at most 2 variables, as in Figure 3.2c. Specifically, TRIDENT infers this specification from the

```
int f(int x, int y, bool ok)
{
        int r;
        r = x + 1;
        if (x < y)
            r = y + 1;
        if(ok > 0) return r;
        else return x+r;
}
```

(a) Low cost patch.

```
int f(int x, int y, bool ok)
{
        int r;
        r = x + 1;
        if (x < y)
            r = ++ok;
        if(ok > 0) return r;
        else return x+r;
}
```

(b) High cost patch.

Figure 3.3: Patches with different cost.

test-passing path:

$$
\begin{aligned}
&(s_x \rightarrow x' = \alpha_x \wedge \neg s_x \rightarrow x' = 0) \\
\wedge\ &(s_y \rightarrow y' = \alpha_y \wedge \neg s_y \rightarrow y' = 0) \\
\wedge\ &\mathit{AtMost}(2, s_x, s_y) \\
\wedge\ &(x' > 0 \wedge y' > 0)
\end{aligned}
$$

After inferring this patch synthesis specification, TRIDENT combines it with with `inc_if_zero`'s summary, if the two are consistent, to synthesise a call to `inc_if_zero(&x,&y)`, which replaces the pair of assignments in Figure 3.2c.

3.2.4 Resisting Overfitting

As demonstrated in Section 3.5.3, the inclusion of patches with side effects into the program repair search space increases the probability of generating test-overfitting patches. To alleviate this problem, we propose a patch prioritization strategy that assigns lower cost to patches that have fewer side effects.

Consider the code fragments in Figure 3.3a and Figure 3.3b. The function $f(x,y,ok)$ is expected to return $\max(x,y)+1$ if $(ok > 0)$, otherwise should return $\max(x,y)+x+1$. Assume that TRIDENT generated patches that inserted the highlighted assignments to pass the test $\texttt{f}(-3,1,1) \rightarrow 2$. Although both programs pass the test, the patch in Figure 3.3b is incorrect, as it breaks the functionality for inputs satisfying $(y > x) \wedge (ok = 0)$.

To reduce the chance of generating such overfitted patches, we propose a

heuristic to minimise the number of side effects in a patch. This heuristic is based on the intuition that updating fewer variables decreases the chance of breaking the functionality that is not covered by the tests. Specifically, the patch in Figure 3.3a is assigned cost 1, since it only changes r. The patch in Figure 3.3b is assigned cost 2, since it changes r and ok. Therefore, TRIDENT prefers the patch in Figure 3.3a due to its lower cost. Although this method does not guarantee that the chosen patch is correct, our evaluation in Section 3.5.3 shows that it does, in practice, alleviate test-overfitting associated with side effects.

3.3 TRIDENT

TRIDENT relies on *symbolic execution* as defined in Definition 7. TRIDENT uses tests as correctness criteria, where a *test* is a pair $([a_1, ..., a_n], \phi)$, where $a_1, ..., a_n$ is a sequence of integer inputs, and ϕ is a predicate on the return value (or symbolic term).

Section 3.3.1 defines the definitions required for the chapter. The next three sections are devoted to the core contributions of TRIDENT. Section 3.3.2 describes the state-merging strategy for alleviating path explosion when inferring specification for patch synthesis. Section 3.3.3 defines a component based synthesis approach that explicitly reasons about side effects. Finally, Section 3.3.4 introduces a patch prioritization strategy for alleviating test-overfitting due to side effects.

3.3.1 Definitions

Our approach is designed for imperative languages that use assignment statements to update program state. This chapter builds on top of the definitions discussed in 2.2.3. To precisely define the semantics of assignment statements, this section employs the notion of value category as in CPL [54]:

Definition 9 (Value categories[1])**.** *An expression in a program is an* rvalue *when it is evaluated in "right-hand mode",* i.e. *appears in a condition or on the right-hand side of an assignment. An expression is an* lvalue *when it is evaluated in "left-hand*

[1]Value categories are defined as in CPL, an ancestor of C and C++, as opposed to the more complex categories in C and C++, to simplify the description of the approach.

mode", i.e. *appears on the left-hand side of an assignment.*

This chapter assumes that all expressions can be evaluated in "right-hand mode", but only certain expressions can be evaluated in "left-hand mode". For example, `x` can be both an lvalue and an rvalue, but `x+1` can only be an rvalue.

A *patch* is a pair of programs (p, p'). The *difference* between p and p' — $diff(p, p')$ — is the minimal (in terms of the number of AST nodes) substitution $\{ s_1^p \mapsto s_1^{p'}, ..., s_n^p \mapsto s_n^{p'} \}$ of statements/expressions in p to statements/expressions in p' such that p' is obtained by simultaneously applying this substitution to p (we assume that s_i^p and $s_i^{p'}$ are all unique statements at different program locations). We call a pair $(s_i^p, s_i^{p'})$ in this mapping an *atomic substitution*. An atomic substitution has a side effect iff there exists a memory, stack pair that defines a context in which the execution of p and p' results in a different value for at least one memory location.

Definition 10 (Atomic Substitution with Side Effect)**.** *An atomic substitution* $(s_i^p, s_i^{p'})$ *has a side effect iff there exists an address* $a \in \mathbb{N}$*, a memory* μ*, and a stack* σ *such that* $\langle s_i^p, \mu, \sigma \rangle \Downarrow \langle \mu_1, \_, \_ \rangle$*,* $\langle s_i^{p'}, \mu, \sigma \rangle \Downarrow \langle \mu_2, \_, \_ \rangle$ *and* $\mu_1(a) \neq \mu_2(a)$*.*

This definition considers side effects syntactically *w.r.t.* the *minimal* difference between a program and a patched version under a given *diff* algorithm.

Definition 11 (Patch with Side Effect)**.** *A patch has a side effect iff an atomic substitution of its difference has a side effect.*

For example, consider a patch

$$(int\ f(y)\ \{\ x = y - 1\}, int\ f(y)\ \{\ x = y + 1\}).$$

The difference of this patch is the minimal substitution $\{y - 1 \mapsto y + 1\}$. The expressions $y - 1$ and $y + 1$ do not write to memory, so their evaluation results in the same memory values for any initial memory and stack. Therefore, this patch has no side effects.

Assume that $\{\alpha_1 \mapsto n_1, ..., \alpha_k \mapsto n_k\}$ is an assignment of the variables from $\mathscr{L}$ (defined in Figure 2.3 We say that this assignment *satisfies* a formula π iff

KHOLE-ASSIGN

$$\frac{((\_,\gamma),\_) = \text{pop}(\sigma) \qquad n = |\text{Dom}(\gamma)| \qquad s_1,...,s_n = \text{fresh bool symb. variables} \qquad \alpha_1,...,\alpha_n = \text{fresh int symb. variables}}{\theta' = \theta[\gamma(v_i) \mapsto \alpha_i]_{v_i \in \text{Dom}(\gamma)} \qquad \pi' = \pi \wedge (\bigwedge_{v_i \in \text{Dom}(\gamma)} \neg s_i \rightarrow \alpha_i = \theta(\gamma(v_i))) \wedge AtMost(k, \{s_1,...,s_n\})}{\langle \square_1^L,...,\square_k^L = \square_1^R,...,\square_k^R, \theta, \sigma, \pi \rangle \Downarrow_m \langle \theta', \sigma, \pi' \rangle}$$

Figure 3.4: Semantics of multi-path specification inference.

a substitution of the variables α_i with the corresponding values n_i (denoted as $\pi[\alpha_1 \mapsto n_1,...,\alpha_k \mapsto n_k]$) evaluates to *True*. We also introduce a concretization of symbolic states defined as follows:

Definition 12 (Concretization)**.** *Let θ be a symbolic state, $\{\alpha_1 \mapsto n_1,...,\alpha_k \mapsto n_k\}$ be an assignment of the variables from L_Σ. A* concretization $\theta[\alpha_1 \mapsto n_1,...,\alpha_k \mapsto n_k]$ *of θ with the assignment $\{\alpha_1 \mapsto n_1,...,\alpha_k \mapsto n_k\}$ is defined as follows:*

$$\theta[\alpha_1 \mapsto n_1,...,\alpha_k \mapsto n_k] \coloneqq \lambda a.\ \theta(a)[\alpha_1 \mapsto n_1,...,\alpha_k \mapsto n_k]$$

that is as a concrete program state (a mapping of memory addresses into values expressed using lambda notation) computed by substituting all the variables α_i in the logical terms in the codomain of θ with the corresponding values n_i.

3.3.2 TRIDENT's Multi-Path Specification Inference

A key part of TRIDENT repair algorithm is inferring specification for patch synthesis. This specification is a logical formula that summarises how changes at the given location can affect the output of the program. The main difference between TRIDENT and previous semantic algorithms, such as Angelix, is that TRIDENT's specification encodes the effect of assigning multiple memory locations. A naïve way to implement specification inference shown in Figure 3.1c causes path explosion and therefore reduces the scalability of program repair as demonstrated in Section 3.5. Thus, this chapter proposes a multi-path specification inference approach that uses a specialised state merging strategy to reduce the number of explored paths.

In order to formalise multi-path specification inference, we extend language $\mathscr{L}$ with a k-holed assignment statement defined below:

Definition 13 (k-holed assignment)**.** *Let k,l be integers such that $l \leq k$, $\square_1^L,...,\square_k^L$ be lvalue-holes, $\square_1^R,...,\square_k^R$ be rvalue-holes, $x_1,...,x_l$ be a sequence of lvalue expressions, $e_1,...,e_l$ be a sequence of rvalue expressions. The semantics of* k-holed assignment *$\square_1^L,...,\square_k^L = \square_1^R,...,\square_k^R$ w.r.t. the substitution $x_1,...,x_l$, $e_1,...,e_l$ is defined as*

$$
\begin{aligned}
&t_1 = x_1\\
&...\\
&t_l = x_l\\
&x_1 = e_1[x_1 \mapsto t_1,...,x_l \mapsto t_l]\\
&...\\
&x_l = e_l[x_1 \mapsto t_1,...,x_l \mapsto t_l]
\end{aligned}
$$

where $t_1,...,t_l$ are fresh variables, and $e_i[x_1 \mapsto t_1,...,x_l \mapsto t_l]$ is the expression obtained by replacing $x_1,...,x_l$ with $t_1,...,t_l$ correspondingly.

The fresh variables $t_1,...,t_l$ in the definition prevent the preceding assignments from affecting the result of the following ones. We refer to this extended language with k-holed assignments as $\mathscr{L}'$.

Multi-path specification inference is defined for programs from $\mathscr{L}'$ using an augmented version of symbolic execution. For simplicity, we assume that there is only a single test, but all definitions in this chapter can be trivially generalised for multiple tests by considering the conjunction of constraints corresponding to these tests.

Definition 14 (Multi-path specification inference)**.** *Let $p \in \mathscr{L}'$ be a program, f be a function declared in p (entry function), $([a_1,...,a_n],\phi)$ be a test. The function infer is defined as $infer(p,f,[a_1,...,a_n]) \stackrel{def}{=} symex_m(p,f,[a_1,...,a_n])$ where $symex_m$ is given in Definition 7 with the semantics $\Downarrow_m$, where the relation $\Downarrow_m$ is an extension of $\Downarrow_s$ to $\mathscr{L}'$ in Figure 3.4.*

3.3.3 TRIDENT's Patch Synthesis

TRIDENT employs component-based program synthesis that constructs a patch as a combination of components that satisfies a given logical specification. The key novelty relative to previous techniques [48, 49] is that it supports side-effected components. Specifically, we consider components of three types identified as sets of labels: $\{R\}$ for components that represent rvalue-expression, $\{L\}$ for components that represent lvalue expressions, and $\{R,L\}$ for components that are both rvalue expressions and lvalue expressions, such as program variables.

Definition 15 (Component)**.** *Component is a tuple* $(T, \{i_1^R, ..., i_n^R\}, \{i_1^L, ..., i_m^L\}, \phi)$*, where* T *is the type of the component (i.e.* $\{R\}$*,* $\{L\}$*, or* $\{R,L\}$*),* $\{i_1^R, ..., i_n^R\}$ *is the set of rvalue inputs,* $\{i_1^L, ..., i_m^L\}$ *is the set of lvalue inputs, and* ϕ *is the semantics of the component, a logical formula over the variables* $\{i_1^R, ..., i_n^R, i_1^L, ..., i_m^L\}$*, representing the input, and* $\{o^t\}_{t \in T}$*, representing the outputs.*

For example, a component that adds one to a given value can be represented as follows

$$(R, \{i_1^R\}, \{\}, o^R = i_1^R + 1),$$

because it is component that has one rvalue input and is itself an rvalue. Meanwhile, a component that increments a variable, is represented as

$$(R, \{i_1^R\}, \{i_1^L\}, o^R = i_1^L \wedge i_1^L = i_1^R + 1),$$

because it accepts an argument as both an rvalue (for reading) and an lvalue (for writing), updates the value of the lvalue input, and returns this value as rvalue output.

This flexible component model allows the representation of a wide range of operations, from simple arithmetic expressions to more complex constructs, such as function calls, by treating them as components within the synthesis framework. As a result, function calls are naturally incorporated into the synthesis process without requiring any special handling, as they are treated like any other component within the unified representation.

Algorithm 1: Patch synthesis

```
Input:
Components: A list of components
S: inferred specification

for tree ∈ enumerate_trees(Components) do
    ϕ = encode(tree, S)
    is_sat, valuation = solve(ϕ)
    if is_sat then
        return decode(tree, valuation)
```

TRIDENT represents patches as trees of components. Specifically, a component tree is a pair $(c, \{i_1^R \mapsto t_1^R, ..., i_n^R \mapsto t_n^R, i_1^L \mapsto t_1^L, ..., i_m^L \mapsto t_m^L\})$ of the root component c and a mapping from its inputs to subtrees $t_1^R, ..., t_n^R, t_1^L, ..., t_m^L$. The semantics of the tree is a formula that connects input and outputs of the components:

$$\begin{aligned}&[\![(c, \{i_1^R \mapsto t_1^R, ..., i_n^R \mapsto t_n^R, i_1^L \mapsto t_1^L, ..., i_m^L \mapsto t_m^L\}]\!] \stackrel{\text{def}}{=} \\ &\phi \wedge i_1^R = o_1^R \wedge ... \wedge i_n^R = o_n^R \wedge i_1^L = o_1^L \wedge ... \wedge i_m^L = o_m^L \\ &\wedge [\![t_1^R]\!] \wedge ... \wedge [\![t_n^R]\!] \wedge [\![t_1^L]\!] \wedge ... \wedge [\![t_m^L]\!],\end{aligned}$$

where ϕ is the semantics of the component c and $o_1^R, ...$ are the outputs of the root components of the subtrees $t_1^R, ...$

TRIDENT uses an enumerative algorithm to find a patch that satisfies the given specification. However, an enumerative algorithm is not efficient for generating integer constants because of search space explosion. To address this, TRIDENT applies an SMT solver to generate these constants, similarly to SyGuS solvers [49]. For example, to represent a set of components that add different constants to a given value, instead of considering concrete semantics $o^R = i_1^R + 1$, $o^R = i_1^R + 2, ...$, we consider an abstract semantics $o^R = i_1^R + c$ and ask an SMT solver to find a value of the parameter c that satisfies the specification.

Algorithm 1 demonstrates the patch synthesis algorithm that combines enumeration and SMT-solving. First, this algorithm enumerates abstract trees, that is, trees of components in which the leafs corresponding to constants are represented as

Algorithm 2: Patch Prioritization

```
1 Input:
2 Patch: A list of components
3 Line: Line number of patch
4 P: Program
5
6 if has_no_side_effects(Patch) then
7     return 0
8 P' := apply_patch(P, Patch)
9 return mutated_memory_count(P', Line)
```

parameters. Then, it encodes each abstract tree and the specification into a formula that is satisfiable iff there is a assignment of constants that makes the patch satisfy the specification. Particularly, for a program p with a k-holed assignment and the entry function f, a test $([a_1,...,a_n],\phi)$, a specification $\{(\pi_{i,},\psi_i)\}_i \stackrel{\text{def}}{=} infer(p,f,[a_1,...,a_n])$, and a candidate components tree t, it constructs the following formula:

$$[\![t]\!] \wedge \bigvee_i \pi_i \wedge \phi(\psi_i)$$

For multiple tests, TRIDENT considers the conjunction of the corresponding formulas. If the formula is satisfiable, then TRIDENT constructs a concrete patch from the model by substituting constant parameters with concrete values.

3.3.4 TRIDENT's Patch Prioritization

TRIDENT employs patch prioritization strategy to alleviate test-overfitting. TRIDENT prioritizes patches based on the assumption that minimising the number of side effects in the patched expression will reduce overfitting. Algorithm 2 returns the patch priority where the patches with lower priority value are preferred by the patch prioritisation strategy. The algorithm takes as input the program, patch and the line number where the patch is applied. The function *apply_patch* applies the patch on program P. The function *mutated_memory_count* returns the number of memory locations whose values are changed by the execution of the patched line *Line* in the program P'. If the patch P has no side effects under Definition 11, then algorithm 2 gives it a priority of 0, to avoid synthesising patches with side effects when they are

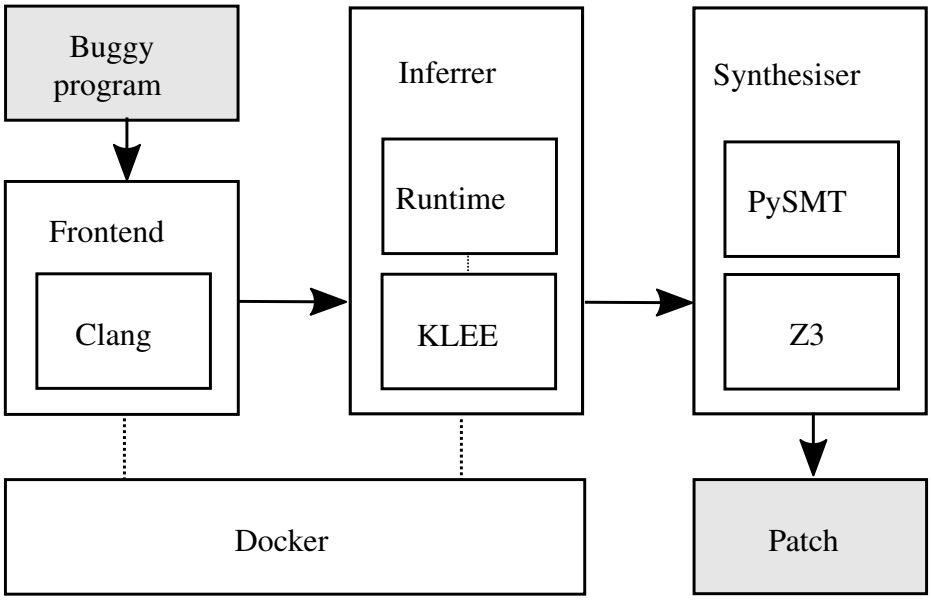


Figure 3.5: Architecture of TRIDENT.

not required.

3.4 Implementation

Figure 3.5 shows the architecture of TRIDENT, which consists of three main components:

- The frontend transforms and analyses buggy programs;
- The inference engine infers synthesis specifications; and
- The synthesiser constructs patches.

The frontend performs several source code focused tasks. First, it localises suspicious locations in the buggy program using Ochiai statistical fault localisation [18]. Second, it instruments suspicious locations by inserting holes in the form of calls to the function `__trident_khole_assignent` using Clang, LLVM's default frontend[55]. Finally, it calls the other components of TRIDENT to infer synthesis specification and synthesize patches.

The inference engine is built on top of KLEE symbolic execution engine [56] and extensions implemented as a C library that is linked to the

buggy program when it is executed using KLEE. The runtime provides a function `__trident_khole_assignent` that represents a k-holed assignment $\square_1^L,...,\square_k^L = \square_1^R,...,\square_k^R$. This function takes the values of program variables and addresses of assignable memory locations, and constructs path constraints and symbolic state according to the semantics of the rule KHOLE-ASSIGN in Figure 3.4.

The synthesizer constructs patches based on inferred specification in the form of KLEE path conditions and provided components and function summaries. It uses PySMT [57] to manipulate SMT formulas, and Z3 for constraint solving.

TRIDENT relies on Docker[58] virtual environments to execute the subject program for fault localization and patch validation, and to perform symbolic execution with KLEE.

The following transformation schemas for C programs, which adapt transformation schema successfully used in previous work [4], define TRIDENT's search space:

$$\langle \text{stmt} \rangle \texttt{ ; } \mapsto \langle \text{stmt} \rangle \texttt{ ; } \square_1^L,...,\square_n^L = \square_1^R,...,\square_n^R \texttt{ ;}$$

$$\texttt{if } (\langle \text{expr} \rangle) \ \langle \text{stmt} \rangle \mapsto c,\square_2^L,...,\square_n^L = \square_1^R,...,\square_n^R \texttt{ ; if } (c) \ \langle \text{stmt} \rangle$$

$$\texttt{while } (\langle \text{expr} \rangle) \ \langle \text{stmt} \rangle \mapsto \texttt{while } (c) \ \{ \ \langle \text{stmt} \rangle \texttt{ ; } c,\square_2^L,...,\square_n^L = \square_1^R,...,\square_n^R \}$$

$$\texttt{for}(\_;\langle \text{expr} \rangle;\_) \ \langle \text{stmt} \rangle \mapsto \texttt{for } (\_;c;\_)\{\langle \text{stmt} \rangle \texttt{ ; } c,\square_2^L,...,\square_n^L = \square_1^R,...,\square_n^R \ \}$$

$$\texttt{switch } (\langle \textit{expr} \rangle) \ \langle \text{stmt} \rangle \mapsto c,\square_2^L,...,\square_n^L = \square_1^R,...,\square_n^R \texttt{ ; switch } (c) \ \langle \text{stmt} \rangle$$

$$\langle \text{call} \rangle \texttt{ ; } \mapsto \square_1^L,...,\square_n^L = \square_1^R,...,\square_n^R \texttt{ ;}$$

$$\langle \text{assignment} \rangle \texttt{ ; } \mapsto \square_1^L,...,\square_n^L = \square_1^R,...,\square_n^R \texttt{ ;}$$

In these transformations, we synthesize a k-holed assignments $c,\square_2^L,...,\square_k^L = \square_1^R,...,\square_k^R$ with a dedicated variable c that is used as an rvalue expressions for conditional statements, loops and switch statements. Effectively, this emulates synthesis of expressions with side effects.

A total of 37 components, including 15 function summaries are constructed offline using symbolic execution.

3.4.1 Limitations

TRIDENT inherits the usual limitations of symbolic execution approaches. First, it faces the usual path explosion problem, which the state merging algorithm, described in Section 3.3.2, alleviates. Second, state-of-the-art symbolic execution engines cannot automatically model the environment, like network communication. Third, SMT solver are necessarily incomplete and cannot solve all expressions in actual programs. Finally, handling pointers and data structures is an open challenge for symbolic execution.

Aliasing occurs when the same data can be accessed through different pointers. Our synthesis algorithm assumes that there is no aliasing in the considered lvalues. Specifically, all lvalues passed to the synthesizer must refer to different memory locations in the context of given tests. TRIDENT's implementation trivially guarantees this property since it only passes references to local and global variables as lvalues. To support dynamic data structures, such as linked lists, an alias analysis technique can be applied. Supporting dynamic data structures also requires modelling them symbolically. TRIDENT's implementation does not support dynamic data structures, because KLEE does not explicitly model them.

TRIDENT assumes that all components are readily available, which may not always be a realistic assumption. Some components, such as function calls, require offline synthesis. Additionally, the components constructed using symbolic execution may lack precision due to the limitations of symbolic execution. This causes the function call component to not accurately represent its actual code counterpart.

3.5 Evaluation

To evaluate TRIDENT, we first demonstrate its utility: we show that TRIDENT can synthesise patches with side-effects for bugs in real-world programs. Synthesising side-effected patches exacerbates two seminal challenges: the path explosion problem of symbolic execution and the test-overfitting problem of program repair. TRIDENT combats path explosion with multi-path specification inference introduced in Section 3.3.2. Section 3.5.2 reports the effectiveness of this countermeasure. Sec-

Table 3.1: CF110 dataset of bugs from Codeflaws benchmark

Class	Number	Description
SISA	38	Insert assignment
DRWV	14	Replace variable with variable
DMAA	3	Insert/replace array access
ORRN	21	Replace relational operator
OILN	15	Tighten/loosen condition
OAIS	19	Insert/delete arithmetic operator
Total	110	

Table 3.2: OSS10 dataset of bugs from open source projects

Program	Commit	kLoC	Missing/incorrect statements
grep	191a84a	38	`match_lines=match_words=0;`
grep	7585d81	27	`strip_trailing_slashes(optarg);`
coreutils	c160afe	208	`x.preserve_xattr = true;`
coreutils	9944e47	249	`if (!nfiles) fstatus[0].failed=1;`
coreutils	ca99c52	249	`if (line_width<0) line_width=0;`
coreutils	9f5aa48	224	`relative_to = relative_base;`
coreutils	2a80912	247	`f[i].fd = -1;`
busybox	0506e29	297	`end = key->range[3];`
busybox	5c13ab4	331	`flags = option_mask32;`
busybox	6f7a009	351	`xtc &= ~TC_UOPPOST;`

tion 3.5.3 shows how much more prone to test-overfitting patches with side effects are than side-effect-free patches, and how well TRIDENT's prioritisation heuristic alleviates this problem.

Benchmarks To answer our research questions, we constructed three bug datasets:

OSS10 10 bugs from open source projects that require patches containing addition/-modification of assignments and function calls, extracted from Coreutils, Grep and Busybox (Table 3.2);

CF110 110 bugs from Codeflaws [52]. Among these 110 versions, 55 require patches with side effects, and 55 require patches without side effects.

MB37 37 bugs taken from ManyBugs [51].

To construct OSS10, we identified bugs from Coreutils, Grep and Busybox that require patches with side effects. We chose these projects for our dataset, because they are well-supported by KLEE, and also their version control history links bug

fixing commits with corresponding regression tests. Specifically, we uniformly sampled bugs from these projects, keeping the first 10 that manual assessment determined involved statements with side effects, those that add/modify assignments or function call. Table 3.2 lists the size of the codebase from which each bug is drawn.

ManyBugs [51] benchmark consists of 185 defects taken from nine large, open source C projects. This benchmark is commonly used in evaluating automatic program repair tools [3, 4, 59]. Prior work [60] argues for explicitly defining the defect classes while evaluating various program repair tools, to ensure a fair comparison of tools on comparable classes. A defect class is a family of bugs that share a common attribute. For instance, GenProg [45] does not repair expression-level bugs, while Angelix [4] does not fix bugs pertaining to insertion of new statements or modifying existing statements in a way that induces side effects. TRIDENT supports the defect classes of Angelix[2] while additionally supporting those defect classes with side effects that k-holed assignment can model. In case when these tools are run on bugs outside their defect classes, these tools will not be able to fix these bugs, although, it could be possible for these tools to generate a patch which passes the tests but is incorrect.

Following previous work [59], we eliminated the samples that do not belong to TRIDENT's defect classes or that TRIDENT could not compiled due to version incompatibilities; this led to MB37, which is a dataset of 37 samples.

We chose Codeflaws as the source for our second dataset, because it contains bugs from a large variety of defect classes. Codeflaws bugs were not labeled with side effects in mind. To construct CF110, we therefore inspected bugs in defect classes that require, or rule out, patches with side effects, and then uniformly sampled bugs from the inspected set. Specifically, we selected 55 bugs from classes SISA, DRWV and DMAA (with side effects), and 55 bugs from classes ORRN, OILN and OAIS (without side effects). Section 7.2 details the reasons for constructing new

[2]Although TRIDENT supports Angelix' defect class, their search spaces are different: Angelix attempts to minimally modify existing expressions, while TRIDENT synthesises expressions from scratch. This explains differences in the results.

```
int clamp(int x, int l, int h) {
    int r;
    klee_open_merge()
    switch(PATCH_ID) {
        case 0: x = □^R; break;
        case 1: l = □^R; break;
        case 2: h = □^R; break;
        case 3: r = □^R; break;
    }
    klee_close_merge()
    if (x < l)
        r = l;
    if (r == x && x > h)
        r = h;
  return r;
}
```

Figure 3.6: Applying KLEE state merging in AKP.

datasets.

Tool Configurations In our experiments, we use the following tool configurations:

TN TRIDENT with disabled patch prioritisation;

TP TRIDENT with enabled patch prioritisation;

PR Prophet [3] with default configuration;

SOS SOSRepair [59] with default configuration;

AKN Angelix-like assignment synthesis with disabled KLEE merging;

AKP Angelix-like assignment synthesis with enabled KLEE merging;

ANG Angelix [4] with default configuration; and

GP GenProg [45] with default configuration.

The TP and TN configurations employ multi-path specification inference described in Section 3.3.2 with cardinality constraints with $K = 2$. AKN is an application of Angelix for assignment synthesis that uses a switch statement to enumerate possibly writable memory locations (Section 3.2.2). AKP is an extension of AKN that applies KLEE's built-in state-merging mechanism by surrounding the switch statement with `klee_open_merge()` and `klee_close_merge()` as shown

Table 3.3: Generated patches for OSS10: ● indicates correct patch, ◐ — plausible patch, ○ — no patch found.

Bug	Patch			Time (s)		
	TP	AKP	GP	TP	AKP	GP
191a84a	○	○	○	1273.2	Timeout	Timeout
7585d81	●	●	○	469.8	554.2	Timeout
c160afe	●	○	○	94.6	Timeout	Timeout
9944e47	◐	○	○	76.3	Timeout	Timeout
ca99c52	○	○	○	Timeout	Timeout	Timeout
9f5aa48	○	○	○	Timeout	Timeout	Timeout
2a80912	●	○	○	75.6	Timeout	Timeout
0506e29	○	○	○	Timeout	Timeout	Timeout
5c13ab4	●	●	○	367.9	348.3	Timeout
6f7a009	◐	◐	○	283.4	349.6	Timeout
Overall	4+2	2+1	0+0			

in Figure 3.6. ANG is Angelix [4] applied only to side-effect-free expressions, but re-implemented using the same synthesiser as TRIDENT with only rvalue components. We used GenProg [45] in our experiments because, although it does not synthesise patches with side effects, it can potentially generate them by copying from other parts of the same program. PR is the original version of Prophet [3] with the default configuration specified in their replication package. For SOS, GP, and ANG we used the generated patches listed in their replication packages to compile the results.

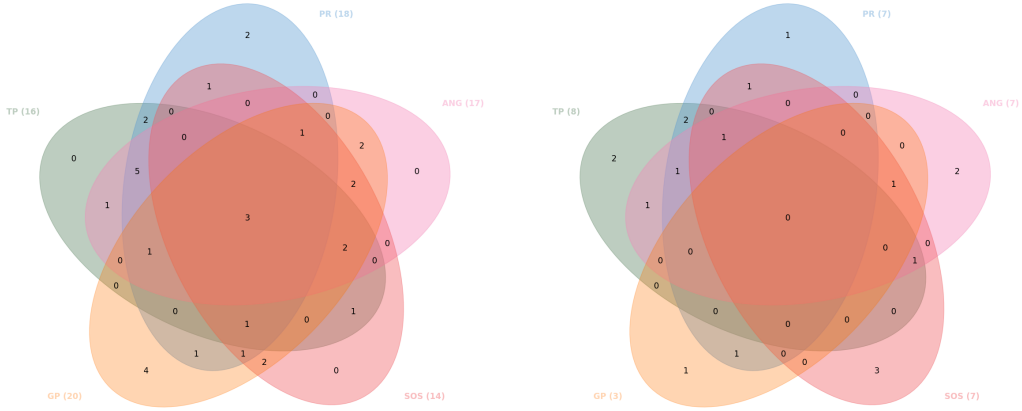
Experimental Setup We conducted all experiments inside a Docker container on an Intel® Core™ i7-2600 CPU 2.7 GHz machine running on Ubuntu 16.04 with 16GB of memory. We used 2 hours as the timeout for each tool.

3.5.1 TRIDENT Fixes Real Bugs

To investigate the applicability of TRIDENT for realistic projects, we ran it on the real bugs in the OSS10 and MB37 datasets. On OSS10 that contain bugs involving side effects, we executed three configurations: TP, AKP and GP. Here AKP serves as the baseline approach. GP is an alternative approach, because it cannot synthesise new statements, but can copy/move them from a code bank, by default the rest of the same program. On OSS10 that contain both bugs involving side effects and side-effect-free bugs, apart from TP, we executed four configurations that represent state of the art C program repair tools: ANG, PR, SOS and GP.

Table 3.4: The number of plausible and correct patches each program repair tool generates on bugs in MB37 dataset. TRIDENT generates 16 out of 37 patches of which 8 are equivalent to patches written by the developers.

Bug	kLoC	Total	Plausible					Correct				
			TN	ANG	PR	SOS	GP	TN	ANG	PR	SOS	GP
gmp	145	2	1	2	1	0	1	0	0	0	0	0
gzip	491	4	3	3	2	0	1	2	1	2	0	0
libtiff	77	10	5	5	5	8	7	3	3	1	2	2
php	1,099	19	6	6	9	4	9	3	3	4	3	0
wireshark	2,814	2	1	1	1	2	2	0	0	0	2	0
Overall		37	16	17	18	14	20	8	7	7	7	2



(a) The number of plausible patches generated by each tool

(b) The number of correct patches generated by each tool

Figure 3.7: Venn diagrams describing the intersection of the repaired bugs in MB37 dataset by different repair tools. Generally, the tools are complementary. TRIDENT correctly fixes 2 bugs that other tools do not repair correctly

In order to identify if the generated patches are correct, we conservatively compared them with human patches, classifying a patch as correct only if it is syntactically identical to the human patch, or can be obtained from the human patch through a trivial refactoring.

Table 3.3 summarises the results of our experiment on OSS10. The time column indicates the time(in seconds) taken to generate a patch for the corresponding buggy location. TP generated more patches than AKP and GP on the considered dataset. TP generated more patches than AKP due to the efficiency of its state merging algorithm. GP could not generate patches for the considered bugs, because required statements are not present in the source code. TP repaired all versions, except for Grep 7585d81,

```
...
  case EXCLUDE_DIRECTORY_OPTION:
    if (!excluded_directory_patterns)
      excluded_directory_patterns =
          new_exclude ();
      strip_trailing_slashes (optarg);
      add_exclude (
          excluded_directory_patterns,
          optarg, EXCLUDE_WILDCARDS);
      break;
...
```

(a) Function call generated by TRIDENT for Grep.

```
...
  if (!TIFFFillStrip(tif,strip))
    return((tmsize_t)(-1));
  size = strip_size;

  if ((*tif->tif_decodestrip)(tif,buf,
      size,plane)<=0)
    return((tmsize_t)(-1));
  (*tif->tif_postdecode)(tif,buf,size);
  return(size);
...
```

(b) Assignment statement generated by TRIDENT for Libtiff

Figure 3.8: Examples of patches generated by TRIDENT.

```
* by the compression close+cleanup routines.  But
* be careful not to write stuff if we didn't add data
* in the previous steps as the "rawcc" data may well be
* a previously read tile/strip in mixed read/write mode.
*/
-  if (tif->tif_rawcc > 0 && tif->tif_rawcc != orig_rawcc
+  if ((tif->tif_rawcc > 0)
&& (tif->tif_flags & TIFF_BEENWRITING) != 0
&& !TIFFFlushData1(tif))
{
```

Figure 3.9: Angelix's patch for libtiff-2007-11-02-371336d-865f7b2.

using one or more assignments. For Grep 7585d81, TP generated a function call shown in Figure 3.8a, which is identical to the human patch. In order to enable this function call synthesis, we first generated summaries for all supported functions in Grep, and used them as components for patch synthesis.

Table 3.4 summarises the results of our experiments on MB37 dataset: TRIDENT's performance is comparable to other state of the art tools. Figure 3.7 shows the overlap of generated patches for different tools. TRIDENT generates 2 correct patches that no other tool could repair, Figure 3.7b. Figure 3.8b shows one of these two patches.

Even though TRIDENT supports the defect classes of Angelix, we can see from Figure 3.7b and Figure 3.7a that Angelix synthesises some bugs that TRIDENT cannot and vice-versa. This is due to the difference in their search spaces: Angelix attempts to minimally modify existing expressions, whereas TRIDENT synthesises expressions from scratch. One illustrative example is Figure 3.9. Here, Angelix successfully generates a patch, since it is easy to modify the existing expression to reach the patch

by simply dropping the expression `tif->tif_rawcc != orig_rawcc`. TRIDENT, in contrast, does not generate a patch, since the patch requires an expression with 11 components, which is infeasible due to the vast number of candidate patches that use up to 11 components.

TRIDENT generated two correct patches exclusively, because the correct patches are not in the search space of the other approaches. These correct patches require inserting an assignment. Angelix cannot generate a patch that inserts an assignment. GenProg and Prophet could not generate these patches, since the needed assignments do not appear in the buggy programs. SOSRepair could not generate them because its performance depends on the size and quality of the database of patterns.

> TRIDENT repaired 3 more real bugs from OSS10 dataset (Table 3.3) that require patches with side effects than the baseline, state of the art semantic repair augmented to synthesise assignments. TRIDENT correctly repaired 2 new bugs from MB37 dataset that the other state of the art tools, *i.e.* Prophet, SOSRepair, Angelix and GenProg did not repair.

3.5.2 Containing Path Explosion

Concretely, path explosion manifests itself during a symbolic run in terms of the number of paths visited. To investigate how TRIDENT's state merging mitigates path explosion in our setting, we executed three configurations — TP, AKN, and (3) AKP— on the both OSS10 and CF110 datasets. We compare these three configurations in terms of the average number of paths each visits during k runs over corpus against the time limit of 10 hours.

Table 3.5 summarises the results of our experiments on CF110 dataset, and Table 3.6 summarises the results of our experiments on OSS10 bugs. Both tables display the average number of paths that each approach explores per suspicious location; for CF110, this average is over all versions from a defect class. In both cases, TRIDENT, under its TP configuration, visits fewer paths than either baseline, demonstrating the effectiveness of its state-merging strategy at coping with path explosion in practice. Figure 3.10 shows a violin plot for the distribution of number

Table 3.5: The average number of paths explored in the CF110 dataset. Here, TRIDENT, under TP, visits 8 fewer paths on average than the closest baseline.

Class	Paths (Average)		
	TP	AKP	AKN
SISA	12.87	18.43	26.69
DRWV	9.15	13.62	25.23
DMAA	38.86	76.18	98.24
ORRN	15.50	30.36	56.77
OILN	16.47	24.33	40.68
OAIS	16.88	24.72	31.92
Overall	27.39	35.51	37.32

Table 3.6: The average number of paths explored in the OSS10 dataset. The buggy versions for which TRIDENT, under its TP configuration, generated a patch are **bolded**. On the corpus, TP visits almost 1000 fewer paths than the closest baseline, on average.

Version	Paths (Average)		
	TP	AKP	AKN
191a84a	5.4	13.2	22.5
7585d81	95.0	96.0	96.0
c160afe	10.0	268.0	1602.5
9944e47	22.0	490.5	616.0
ca99c52	5688.0	6024.2	6352.0
9f5aa48	1175.0	1320.0	9000.0
2a80912	8.0	39.0	104.0
0506e29	12922.0	16807.0	16807.0
5c13ab4	88.0	317.0	936.0
6f7a009	12647.0	15966.0	21378.0
Overall	3266.1	4133.9	5691.3

of paths for each class in CF110; here, we see that, for each of TP's runs, the bulk of the area of each violin plot is lower than for the baselines and, crucially, that its tail of outliers is even more markedly lower.

Table 3.7 demonstrates the solving time and the number of solver queries of the patch synthesiser for OSS10. On average, TRIDENT required fewer solver calls, but since the constraints it passes to the solver have additional clauses to control patch side effects, the total solving time does not significantly differ from other techniques. The key advantage of TRIDENT is not in reducing the patch generation time, but reducing the number of paths that are necessary to explore to find a test-passing path, which increases the chance of finding a patch, as demonstrated in Section 3.5.1.

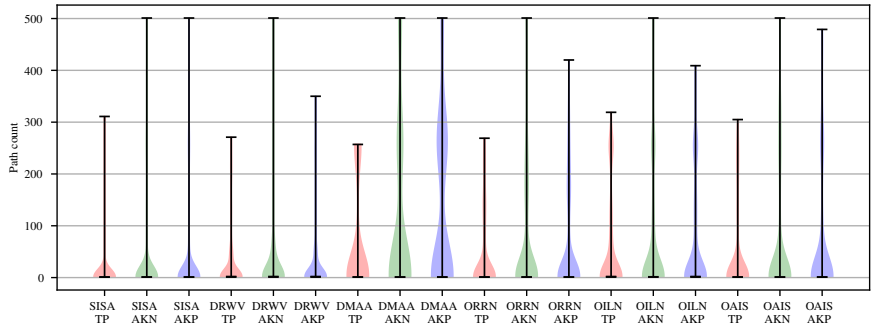


Figure 3.10: The distribution of paths explored in the CF110 dataset. The x-axis contains the defect class name and the name of tool configuration. The weight of each violin plot is lower and their tails are shorter than either baseline.

Table 3.7: The patch synthesizer's solving time and the number of solver queries for each tool for the bugs in OSS10 dataset. The versions for which TP generated a patch exclusively are **bolded**. TP has, on average, fewer solver queries, but since its constraints involve additional clauses, the solving time does not considerably differ across configurations.

Version	Solving Time (Seconds)			Query Count		
	TP	AKP	AKN	TP	AKP	AKN
191a84a	99.0	107.3	76.9	1794	1199	372
7585d81	2.4	2.4	2.1	9	9	9
c160afe	17.3	246.0	164.5	1197	8258	7827
9944e47	21.7	162.4	194.8	1328	6836	1213
ca99c52	241.7	170.6	110.2	184	579	461
9f5aa48	1131.1	661.2	1467.4	1570	3617	7824
2a80912	29.1	94.6	72.2	620	564	561
0506e29	79.3	74.7	70.5	72	64	83
5c13ab4	106.2	173.5	171.1	10873	10034	3206
6f7a009	92.8	95.5	98.1	2833	2985	2523
Total	1820.6	1788.2	2427.8	20480	34145	24079

Overall, TP explores 22.9% fewer paths on average compared to AKP and 26.6% fewer paths when compared to AKN on CF110 dataset, and 21% fewer paths on average compared to AKP and 43% fewer paths when compared to AKN on OSS10 dataset.

TRIDENT's novel state merging strategy reduces the number of explored paths by 22–43% compared to the baselines — state of the art semantic repair augmented to synthesis assignments and KLEE's general-purpose state merging strategy.

Table 3.8: Plausible and correct patches generated for CF110 bugs; as expected, TRIDENT slightly increases test-overfitting on patches without side effects (first three rows) and its patch priorisation strategy reduces test-overfitting, as the comparison between TP and TN on patches with side-effects (the last three rows) shows.

Class	LoC	Plausible			Correct		
		TP	TN	ANG	TP	TN	ANG
OILN	46 ± 21	2	2	1	0	0	0
OAIS	36 ± 25	8	8	7	1	1	1
ORRN	53 ± 36	7	7	7	2	2	3
Overall		17	17	15	3	3	4
SISA	59 ± 35	16	16	1	9	6	0
DRWV	46 ± 20	8	8	0	5	2	0
DMAA	83 ± 28	0	0	0	0	0	0
Overall		24	24	1	14	8	0

3.5.3 Resisting Overfitting

A program repair technique overfits a test suite when it produces patches that pass the test suite, but are incorrect. Following convention, we call patches that pass a test suite *plausible*. We measure the degree of test-overfitting as $1 - \frac{\mathfrak{C}}{\mathfrak{P}}$, one minus the ratio of correct $\mathfrak{C}$ to plausible patches $\mathfrak{P}$. In this experiment, we define correctness as passing the held-out test suite provided by Codeflaws benchmark.

We first establish how extending the search space with side effects exacerbates test-overfitting, then show that TRIDENT produces a higher proportion of correct patches than the baseline. For the former, we compare TRIDENT with ANG, which is only applied to expression without side effects. For the later, we compare the overfitting rate of two TRIDENT's configurations: TP (with enabled patch prioritisation), and TN (with disabled patch prioritisation). We execute all the above configurations on CF110.

Table 3.8 summarises the results. The top three classes OILN, OAIS and ORRN contain programs that can be fixed with a side-effect-free patch and the bottom three classes SISA, DRWV and DMAA require side effects. TP and TN generate more patches than ANG for the side-effect-free classes OILN, OAIS and ORRN but the extra patches fail the held-out tests. TP and TN both have the test-overfitting ratio of 82.4%. Meanwhile, the test-overfitting of ANG is 73.3%. These results indicate that

adding patches with side effects to the search space increases test-overfitting. For classes with side effects SISA, DRWV and DMAA, TP has the test-overfitting rate of 41.7%, and TN has the test-overfitting rate of 66.7%. ANG, in contrast, generates one patch which is incorrect. The results demonstrate that TP's new prioritisation heuristic alleviates test-overfitting by decreasing the rate test-overfitting from 66.7% to 41.7%.

Despite the fact that handling patches with side effects expands the search space thereby increasing the rate of test-overfitting, TRIDENT's specialised patch prioritisation alleviates this negative effect, reducing overfitting from 66.7% to 41.7%.

3.6 Threats to Validity

The results for the tools ANG, SOS, and GP were taken from their respective reproduction packages, as noted in Section 3.5.1. However, the experimental environments and systems used to compute these results differ from those in our setup, which introduces a threat to validity.

Chapter 4

Rete: Learning Namespace Representation for Program Repair

4.1 Introduction

Every program defines a namespace and uses scoping rules to control variable visibility. Existing program repair algorithms neglect information about program's namespace when searching for repairs, which reduces their effectiveness and increases test-overfitting [53]. Patch generation techniques that use machine learning usually fall into three categories: They (1) ignore information about program variables, effectively only learning to choose the template into which to insert variables [3]; (2) only extract variables from local context [29, 61]; or (3) learn information about program namespace implicitly [12], but have difficulty handling long-range dependencies [62]. At each program location, many visible variables are not local, so tools that fall into the first two categories cannot effectively synthesize patches that require non-local visible variables. CoCoNut [12] falls into the third category, so, in principle, it can learn long-range dependencies, but our experiments (Table 4.9) show that it fails to generate five correct patches because it prefers variables in the local neighbourhood to more suitable visible variables.

This chapter proposes RETE, a new program repair technique, to address this problem. RETE prioritises visible variables in a program namespace by their likelihood of being used at a given program location. In the spirit of neuro-symbolic

computation [63], RETE combines program analysis and machine learning to learn rich project specific *semantic* information about the namespace. Specifically, it uses variables' CDU chains, def-use chains augmented with control flow, to learn latent, low-dimensional representations of program variables that capture their semantic affinities.

RETE generates patches by inserting program variables into patch templates. It separates patch template generation and prioritisation into two steps and defines an interface for each step to facilitate the use of different algorithms for each one. Thus, RETE is a framework that integrates existing program repair approaches. RETE prioritises patches by combining the ranks of variables and the ranks of patch templates into which the variables are inserted into a single ranking. Any combination of tools can be used to extract these individual template and variable ranks. We implement three instances of template ranking using existing approaches: (1) the plastic surgery hypothesis [47] that extracts templates from the buggy program, mutates them, and prioritises the mutated templates based on the syntactic distance from the donor code; (2) Prophet's enumerative synthesiser and machine learning based prioritiser [53]; and (3) Trident's constraints-based synthesiser (Section 3.3.3) and combine them with our various variable ranking algorithms (Section 4.3).

We implemented RETE for C and Python. We chose C because of C's importance and because many program repair techniques have targeted it. We target Python because of its ever-increasing importance and the fact that its default dynamic typing heightens the importance of namespace information, since without static types, every visible variable is a candidate for fixing a bug.

To evaluate RETE, we use two benchmarks: 107 bugs extracted from BugsInPy [64] and 35 bugs extracted from ManyBugs [51]. We extracted these bugs to match RETE's defect class described in Section 4.3.1 (with a restriction to inserting/modifying single line statements). The evaluation demonstrates that RETE generates patches for 29 of our 107 BugsInPy bugs, and generates 8 for our 35 bugs ManyBugs. When we adapted Prophet to work on Python, despite its age, it outperformed the previously available state of the art, CoCoNut: fixing 21 bugs

to CoCoNut's 16 bugs on our dataset (section 4.5). On Python, RETE outperforms Prophet-for-Python by six correct fixes. When using RETE's variable prioritisation, Prophet-for-Python generates 3 more bugs than it does on its own. On the ManyBugs dataset, Trident's constraint-based algorithm (Section 3.3.2), augmented with RETE's template enumeration and variable prioritisation, generates more correct patches than the state of the art tools SOSRepair [59], Prophet [53] and Trident (Chapter 3) on their own.

This chapter makes the following contributions:

- We introduce the problem of learning program namespace and present two solutions using deep learning and feature engineering which use information from program analysis.

- We present a generic algorithm to integrate variable prioritisation into existing repair techniques, instantiated for the Plastic Surgery Hypothesis, Prophet and Trident.

- We realise these contributions in RETE, and evaluate them on real bugs in C and Python projects.

RETE's implementation, and the scripts and data used to evaluate it, can be found in the accompanying package `https://github.com/norhh/Rete`

4.2 Overview

To repair real-world bugs, program repair has to explore huge search spaces of candidate patches. Consider the developer patch for a bug in Black [65] in Figure 4.1a. Even if we restrict the number of field accesses with the operator '`.`' to the maximum of two, there are 11 118 visible variables and object field accesses at the fix location. This number can be derived by traversing the tree of object attribute accesses. By limiting the depth of this traversal to two consecutive accesses, we count all possible combinations of variables and their fields that can be accessed within this two-access restriction, resulting in 11 118 variables.

```
if (
      self.previous_line
      and self.previous_line.
            is_decorator
):
    return 0, 0

+ if (
+     is_decorator
+     and self.previous_line
+     and self.previous_line.
       is_comment
+ ):
+    return 0, 0


newlines = 2
if current_line.depth:
    newlines -= 1
```

(a) The developer fix for a bug in Black.

```
if (
    self.previous_line
    and self.previous_line.
          is_decorator
):
    return 0, 0


if □[1] and □[2]:

    return 0, 0


newlines = 2
if current_line.depth:
    newlines -= 1
```

(b) A template for a hypothetical repair tool.

```
if (
    self.previous_line
    and self.previous_line.
          is_decorator
): # Code block used to generate
     fix
    return 0, 0


if □[1] and self.previous_line.□[2]:

    return 0, 0


newlines = 2
if current_line.depth:
    newlines -= 1
```

(c) Edited locations with abstract variables.

Figure 4.1: A bug in Black Python formatter, and how it can be fixed with patch templates.

A popular approach for reducing the search space, employed by a large number of existing program repair tools, is to ignore the variables that are not local to the fix. However, considering only local variables makes some bugs impossible to repair and increases the likelihood of generating test-overfitting patches. For example, the field access `self.previous_line.is_comment` in Figure 4.1a, which is used in the fix, does not exist anywhere else in the entire program. Even assuming that a prioritisation algorithm shortlists 100 candidate variables and we know the exact fix template given in Figure 4.1b, constructing such a patch would require examining 4950 possibilities to fill the variables into the template.

Program repair approaches relying on program synthesis struggle to generate patches for programs with large namespaces. For example, Trident, which was discussed in Chapter 3, enumerates patches using patch templates from its search space and checks whether they satisfy the specification constraints. Applied to the bug in Figure 4.1, for simplicity, let us assume that Trident uses **T** patch templates, Trident must enumerate *ca.* 8–9 orders of magnitude ($\mathbf{T} \times 11118^2$) patches to find this fix, a clearly infeasible number of patches. Prophet [3] and SPR [9] map variables to the values that they hold during test execution. They use this mapping to instantiate templates. If the number of candidate variables is large, this can lead to test-overfitting, since many variables can take the same values during test execution, leaving these tools unable to differentiate them.

RETE efficiently synthesises the correct patch in Figure 4.1a by prioritising the correct patch in the first 1000 patches. To do this, RETE employs a novel method to rank variables using a project-independent representation of a program's namespace.

4.2.1 RETE's Template Generation

RETE's core contribution, learning program namespaces, is orthogonal to existing program repair algorithms. Thus, RETE provides an extensible framework for program repair, which seamlessly integrates existing patch generation algorithms. We consider three archetypal algorithms: the plastic surgery hypothesis [47], Prophet's enumerative synthesiser [53], and Trident's constraints-based synthesiser (Section 3.3.3).

We now show how RETE uses the plastic surgery hypothesis. Our interpretation of the plastic surgery approach starts with existing program statements as partial patches and edits them using one of three operations: replacing a variable with a hole $\square$, appending an operator with holes for its operands, or removing an operator and its operands. The synthesis algorithm searches for a minimal edit patch. The cost of adding or removing an operator equals the number of holes added or removed. Consider the statement $x+a\times b$, the set $\{x+y, a+self.W\}$, and the rewrite chain $x+y \rightarrow_1 x+\square_1 \rightarrow_2 x+\square_1\times\square_2 \rightarrow_3 x+a\times\square_2 \rightarrow_4 x+a\times b$. The chain shows that $x+a\times b$ is 2 edits from the set because the cost of both $\rightarrow_1$ and $\rightarrow_2$ is 1 and the cost of both $\rightarrow_3$ and $\rightarrow_4$ is 0. Section 4.3.4 explains how we set the edit costs. RETE later fills the variable holes with concrete variables using the ranking it learns.

In Figure 4.1c, the code block in lines 1-5 (The if block with the comment) is two edits away from the required patch in Figure 4.1c:

```
1.self.pl              → is_decorator
2.self.pl.is_decorator → self.pl.is_comment
where self.pl represents self.previous_line
```

This patch is correct, if `self.previous_line` is not `None`[1]. As is common practice in APR, RETE guards such deferences with NoneType checks. Given these edits and its NoneType heuristic, RETE generates the human-equivalent patch as shown in Figure 4.1a.

[1]NoneType is Python's unit type.

4.2.2 RETE's Variable Prioritisation

RETE's variable ranking model shortlists the most likely variables at the location of the fix based on a partial patch. This reduces not only the number of variables to examine but also the likelihood of test-overfitting. In Figure 4.1c, the correct variables to fill the holes in "□[1] `and self.previous_line.`□[2]" are ranked 4[th] and 5[th], much less than 782 and 12, the number of visible variables at each hole. The rank for the second variable is not as good as that of the first since no identifier that could fix the problem appeared in a similar context sufficiently often for the model to learn it. Purely ML-based approaches, *e.g.* those using Neural Machine Translation (CoCoNut), struggle to fix such bugs, since the field, `is_comment` is too sparse for the model to learn to associate it with `self.previous_line`.

4.3 RETE

The central "signal" hypothesis of this project is that variables with similar semantics are used in similar ways across code bases; this *usage signal* complements and can supersede signal in raw lexical similarity of names. We introduce conditional def-use chains to capture usage signal.

We consider a C-like programming language $\mathscr{L}$. Program $p \in \mathscr{L}$ is a set of function declarations. In p, let V be its set of variables, and S be the set of all statements in L. Let $\square$ represent a missing variable and $V_\square = V \cup \{\square\}$. The language $\mathscr{L}_\square$ extends $\mathscr{L}$ by replacing V with $V_\square$. Let $\delta \in \Delta$ be a patch; $\delta(p)$ denotes the application of δ to p. $\Delta_\square \subset \Delta$ is a set of patch templates where $\delta_\square(p) \in \mathscr{L}_\square, \forall \delta_\square \in \Delta_\square$. $\delta_\square(\overline{v})$ denotes instantiating $\delta_\square$ with the variables $\overline{v} \in V^n$, where $|\overline{v}| = n$. RETE's defect class [60] consists of those bugs that instantiations of its patch templates can fix.

4.3.1 The Patch Ordering Problem

Generate-and-validate program repair approaches find patches by enumerating and testing a large number of candidate patches. Typically, the first patch found that enables the program to pass the test suite is returned as the solution. Thus, the order in which the patches are enumerated is crucial. First, it affects patch quality

since not all patches that pass the tests are correct. Second, it affects the speed of patch generation since it determines the number of test executions required to find a suitable patch.

Problem 1 (The Patch Ordering Problem)**.** *Given a test suite T, and a program p that does not pass T and a set of patches Δ, find the bijection $O : \Delta \to [1..|\Delta|]$ such that*

$$\operatorname*{argmax}_{O} P\Big(O(\delta_i) < O(\delta_j) \mid \delta_i(p) \textit{ is correct} \\ \vee \big(\delta_i(p) \textit{ is plausible} \wedge \delta_j(p) \textit{ is implausible}\big)\Big),$$

where $\delta(p)$ is plausible if it passes all tests in T and implausible otherwise.

O is a patch ordering that simultaneously maximises the probability that correct patches precede plausible patches, and the probability that plausible patches, in turn, precede implausible patches.

RETE solves a restricted version of this general problem: it optimises O only over instantiated templates $\delta_\square(\bar{v})$. Because of this restriction, RETE decomposes the patch ordering problem into two subproblems: that of ordering templates, aka $\delta_\square$, (Section 4.3.2) and, for each template, the problem of finding the correct variables to fill those holes, *a.k.a.* finding $\bar{v}$ (Section 4.3.3).

4.3.2 Prioritising Templates via Distance

Let $\alpha[vars(\alpha)/\square]$ be the buggy context with holes replacing all its variables, $vars(\alpha)$. We formulate the problem of finding the best template for a particular bug in the program p as a graph search problem, starting from $\alpha_\square$. Let $G = (\Delta_\square \cup \{\alpha_\square\}, E)$ be our graph. The distance function $d\colon E \longrightarrow \mathbb{W}$ weights each $(\delta^s_\square, \delta^t_\square) = e \in E$. Templates are ordered by their distance from $\alpha_\square$.

Different APR approaches define d differently. Section 4.4.1 gives our definition. Techniques such as DirectFix [26] define d as the minimal number of sub-expression substitutions required to construct the patch from a buggy statement. Techniques such as Prophet [3] order templates using Maximum likelihood estimation, implicitly

defining d using probability.

4.3.3 Learning Namespace Representations

In this chapter, we introduce namespace representations learning, which aims to learn latent, low-dimensional representations of program variables, which preserves semantic properties of variable uses, and thus facilitates such applications as variable prioritisation for patch synthesis.

We now describe how we capture a variable's uses in a Conditional DU (CDU) chain, a new data structure that augments a classical DU chain with control information. A variable may have many CDU chains, so we describe how we sample them before describing how we use them to learn embeddings that capture affinities between variables (their names and uses) and holes, each of which represents a potential variable use.

The predicate $D : S \times V$ indicates if a variable is *defined* in a statement; the predicate $U : S \times V$ indicates if a variable is *used* in a statement. For a variable v and a definition d (*i.e.* $D(d,v)$), we say that d reaches an arbitrary use-statement s (*i.e.* $U(s,v)$), if there exists at least one execution path from d to s along which no other statement $s' \neq s$ satisfies $D(s',v)$. A *def-use chain* of the variable v is the sequence of all statements $s_1,...,s_n$ *s.t.* 1) $D(s_1,v) \wedge \bigwedge_{i=2}^{n} U(s_i,v)$; 2) The definition of v in s_1 reaches s_i for all $i > 1$ along at least one path; and 3) $\forall i, j$ *s.t.* $i < j \wedge i \neq 1$, s_i precedes s_j in the source code.

A node d of a graph *b-dominates* a node e if every path starting from a node b to e traverses d. $Dom_b(d)$ denotes the set of all nodes that *b-dominate* the node d. For the statement s in program p, let $(g_1 \ldots g_n)$ be the set of conditional statements in p each of whose elements g_i dominate s. For the sequence $seq = \langle a_1, \ldots, a_n \rangle$, we write $x \in seq$ to denote $\exists i \in \{1, \ldots, n\}\,.\, x = a_i$.

Definition 16 (Conditional Definition-Use (CDU) Chain)**.** *For the variable v, a* conditional def-use chain w.r.t. *an initial node b in a control flow graph is the sequence of statements* $c = \langle s_1, \ldots, s_n \rangle$ *where*

- *A subsequence of c is the DU chain d of the variable v; and*

- *Any statement $s_i \notin d$ is a conditional statement* s.t.

  $\forall s_j \in d\,.\, i < j \implies s_i \in Dom_b(s_j)$.

CDU chains are formed by interleaving a DU chain of v with all conditional statements that b-dominate a statement in the underlying subsequence of the DU chain. In c, the condition g_i precedes the arbitrary element s in c's DU chain if g_i dominates s. Definition 16 non-deterministically orders the conditions *w.r.t.* each other, subject to the constraint that a conditional must precede all statements it dominates. Using b-domination allows us to choose a starting node closer to a variable's uses than a program's entry.

Our intuition is that CDU chains of a given length likely have more information about their variable than arbitrary code snippets of the same length. CDU chains are related to program slices, which consist of all the statements and predicates that might affect a set of variables at a program point [66, 67, 68]. Unlike slices, CDU chains ignore, by design, a variable's data dependencies for two reasons: 1) to avoid the data dependency clusters that bloat many slices [69] and 2) to focus on capturing variable-specific signal.

The *variable prediction task* takes a list of statements with a hole at a variable use and predicts the correct variable to fill that hole. RETE models this task as a masked language modelling problem. Pre-trained, masked language models for code, based on transformers [1], fine-tuned with a small amount of labelled data, achieve state-of-the-art performance in different software engineering applications [70, 71, 72]. Accordingly, RETE adopts this approach to learn affinities between variables and their uses in CDU chains.

To train its variable prioritiser model, RETE repeatedly serialises each CDU chain and masks each use of a variable, not just the defined variable, ignoring all other tokens. For instance, three CDU chains containing five variable uses would generate 15 distinct masked training instances. Figure 4.2 shows us how the model works. For a CDU chain for the variable `a` ending with `a > b`, one instance replaces the final use of `a` with □ to mask it out and then feeds it to a pre-trained large language model to convert the input into a sequence of embeddings. RETE then runs these

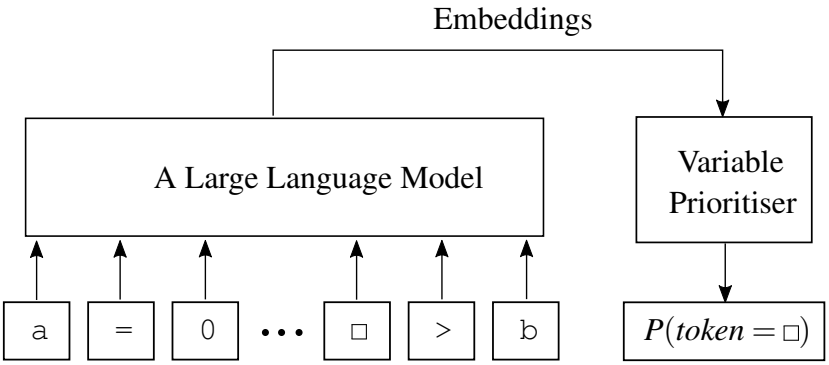


Figure 4.2: Processing a sample CDU chain.

embeddings through a feed-forward network with a softmax layer to realise the task-specific fine-tuning.

Many CDU chains can traverse a given buggy line, so, while its individual CDU chains may be short, the aggregate length of a variable's set of CDU chains can be large. Transformers take a maximum length sequence of tokens as input. Thus, the serialisation of a variable's CDU chains may need to be compressed. Section 4.4.2 details how RETE's framework serialises and compresses CDU chains.

4.3.4 Jointly Prioritising Patches

RETE ranks patches by combining template and variable priorities. RETE uses min priority queue, ordered by Equation (4.1). Equation (4.1) defines $score(\delta_{\square}(\bar{v}))$ using $td(\delta_{\square}) \in \mathbb{N}$ which gives $\delta_{\square}$, the priority of the template and $P(\delta_{\square}, \square, v) \in [0,1]$, the probability of $v \in V$ being the candidate to fill the single hole $\square$. Template distance *td* (Section 4.3.2) calculates the distance of the template from the source node extracted from the localiser (*i.e.* $td(\delta_{\square}) = d(\delta^{S}_{\square}, \delta_{\square})$). The variable prioritiser described in Section 4.3.3 provides variable probabilities. A Lower score means a better patch.

$$score(\delta_{\square}(\bar{v})) = td(\delta_{\square}) + \frac{\theta}{|h(\delta_{\square})|} \sum_{\square_i \in h(\delta_{\square})} \frac{1}{P(\delta_{\square}, \square_i, v_i)} \tag{4.1}$$

where θ is a constant we use to control the growth of the summands, and the function $h(\delta_{\square})$ returns the set of holes in $\delta_{\square}$. We use $\frac{1}{P(\delta_{\square}, \square_i, v_i)}$ in the summation because it gives lower score for variables with a higher probability. As the range of $td(\delta_{\square})$ is in $\mathbb{W}$, various template scores differ by integers, the variable prioritisation score

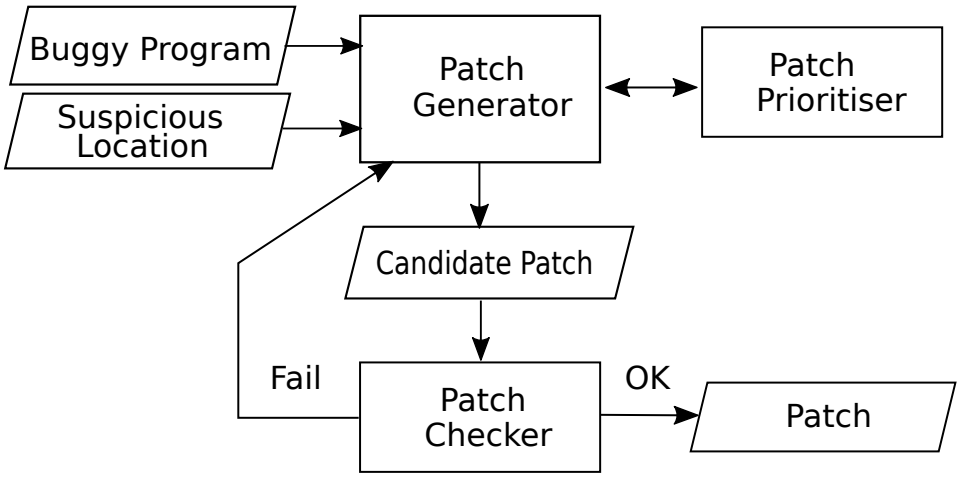


Figure 4.3: Architecture of RETE.

starts to matter when $P(\delta_{\square}, \square_i, v_i) < \theta$, as the score breaks the barrier of 1. We use $\theta = 0.073$ as this value performed the best in our experiments. We prioritise patches by separately prioritising templates and variables rather than directly prioritising instantiated patches because the number of instantiated patches is much larger than the number of templates, making it infeasible in practice.

4.4 RETE's Implementation

Figure 4.3 shows RETE' architecture, which has three main components: (1) Patch Generator, (2) Patch Prioritiser combining Variable Prioritiser and Template Prioritiser, and (3) Patch Checker. RETE's implementation takes in a buggy program p and a suspicious line number and feeds p to its patch prioritiser, which combines its template and variable prioritisers to produce a patch that it checks with its Patch Checker. If the check fails, RETE generates and checks patches until it finds a successful patch. RETE implementation is a framework, so its implementation components are not fixed; they can be instantiated with various existing approaches. For instance, its checker can be anything ranging from a test suite to a patch specification extracted from symbolic execution.

When realised as a generate-and-validate approach, RETE prioritises tests that failed the previous run for efficiency in practice. To combine templates and variable priorities, RETE lazily constructs a priority queue ordered by Equation (4.1).

4.4.1 Lazily Prioritising Templates

To realise RETE's template prioritisation scheme (Section 4.3.2), we need to define the buggy context, our templates, and a distance function for our templates, then use these components to build a weighted graph. To define the buggy context, we use fault localisation. Many different localisers have been used in APR for C. For our C dataset, we use the Ochiai statistical fault localisation [18], since it performs well [73]. Variation in localisation has confounded tool comparison in C. Perhaps, for this reason, prior work on Java APR [74, 75] assumes perfect localisation, as it facilitates tool comparison. CoCoNut is the only previous Python APR work of which we are aware, and it assumes perfect localisation. For our Python dataset, we follow its lead.

We turn to the confirmation of the Plastic Surgery Hypothesis (PSH) [47] to define our templates. PSH shows that many fixes can be constructed from existing code in a codebase. Relying on the PSH, we construct templates starting from atomic statements surrounding the buggy context. We templatise these statements by replacing their variables with holes, one at a time. A statement with three variable uses would generate three distinct templates, each with a single hole. We start with only single-holed templates to preserve signal in the names of the other variables in keeping with RETE's central conceit: the application of Firth's dictum [76] to variables.

The distance between two templates is the difference in the number of holes each has. Thus, we prefer templates with fewer holes, in keeping with PSH, are likely to generate instantiations closer to the buggy statement in edit distance.

Starting from this initial set of templates, RETE's framework uses Dijkstra's algorithm [77] to lazily construct its template graph. First, the framework interconnects the initial templates with zero cost edges to favour their use. It weights subsequent edges by the distance of the templates they connect. It moves to the next template only when it has enumerated all variable instantiations of the current template whose score, Equation (4.1), exceeds the distance of the next template. RETE's framework constructs new templates from existing ones by replacing, appending, or

removing statements. Replacing a statement cannot change the number of holes: it must both fill a hole and replace a variable with a hole. Append and remove change the number of holes, so long as the resulting template has at least one hole, *i.e.* remains a template. When filling a hole, we try each of the variable prioritiser's top 30 variables. This restriction was because maintaining all possible bindings is quite expensive, and we observed that it was unnecessary in practice on our corpus. We hypothesize that this observation generalises. We picked 30 as it effectively balanced cost and performance on our corpus. When creating a template, RETE's framework nondeterministically chooses an operation. We chose the templates surrounding the line of code as the set of initial templates, and we restricted the template size to a single statement and the size of the set to 20 templates.

4.4.2 Variable Prioritisation with CodeBert

We instantiate the pre-trained component of RETE's variable prioritiser (VP) with CodeBERT [78] and fine-tuned CodeBert for C and Python datasets. CodeBERT [78] is a model pre-trained on natural language (NL) and programming language (PL) pairs across six languages: Python, Java, JavaScript, PHP, Ruby, and Go, using 2.1 million NL-PL data points. Since CodeBert is not trained for C, we replaced C-specific symbols, such as *NULL* with 0, and dropped *volatile* quantifiers. We then feed this modified CDU chain to CodeBert to produce embeddings for the task-specific model that fine-tunes CodeBert to RETE's variable prioritisation task: to accurately predict variables for holes in CDU chains. VP's task-specific fine-tuned subcomponent is a feed-forward layer with a softmax function, implemented using Huggingface's open-source transformer [79].

At inference time, we have a buggy statement with a hole. We gather the CDU chains that share this statement, as they comprise the variables that can fill the hole. Under preliminary experimentation, concatenating these chains was inaccurate, so we decided to build a query by interleaving them. To do so, we order statements by reachability, then line number. If one statement is reachable from another in the program's control flow graph, the statement that is reached follows the other in the interleaving; if the statements are mutually reachable or unreachable, we order them

by line number. We perform these actions to maximise variable diversity, as we want to improve the cases where uncommon variables are used to fix the bug by increasing unique variable information in our CDU chains.

To fine-tune the model, we trained the task-specific submodel on interleaved CDU chains so that the training data matches the prediction queries. To produce the shared hole needed for interleaving, we converted each input program into a set of single-holed programs, each with a different variable replaced with a hole. We interleaved the set of CDU chains that pass through each single-holed program's holed statement.

CodeBERT's window size is 512 tokens. In our corpus, 5692 CDU chains pass through a buggy line on average, each having an average length of 34. Naïvely interleaving these produces inputs whose average length is $5692 \times 34 > 512$. To cope with this mismatch, we compress our inputs. First, we consider only cardinality at most five subsets of the CDU chains that pass through a single-holed statement. Then pick that subset whose chains maximise the number of distinct variables that they use, to maximise, as with dropping duplicates above, to increase diverse information on variables. This is the NP hard unweighted maximum coverage problem [80], whose solution we greedily approximate [81]. If required, we next reduce the number of conditions in each chain in the interleaving to the two that use the most variables. If the input is still too long, we truncate it.

Random Forest Variable Prioritiser Several existing program repair techniques [61, 82] use machine learning with feature engineering to tackle test-overfitting. These techniques include features related to program variables, some of which are language-specific and not applicable to C or Python. In the spirit of these techniques, we implemented an alternative approach, based on feature engineering. This approach uses a random forest, and the language-independent features in Table 4.1, to rank program variables. We train this random forest using Scikit-learn [83].

Feature	Description
`lvalue`	count of variable definitions
`rvalue`	count of variable uses
`for_init`	count of **`for`** loop initialisation uses
`for_cond`	count of **`for`** loop condition uses
`for_lcv`	count of loop control uses
`while_cond`	count of **`while`** loop condition uses
`if_cond`	count of **`if`** condition uses
`hole_to_def`	distance between hole and the def
`last_use`	distance to last use
`hole_window`	count of uses in k lines around hole
`operator_histo`	multiset of counts of operator/function uses
`is_global`	local/global variables

Table 4.1: The features that our random forest uses. Each feature is tracked for each variable in scope.

4.5 Evaluation

This evaluation uses many configurations of APR techniques, so it starts by analytically defining APR components, then using those components to establish a new taxonomy. It then turns to describing experiments whose aim is to answer the following questions:

- Do our conditional definition-use (CDU) chains contain strong signal about variable usage? (Section 4.5.2)
- Is RETE's prioritisation strategy effective? (Section 4.5.3)
- Does the best combination of RETE's components advance the state of the art? (Section 4.5.4)

Corpus Our corpus consists of three bug datasets whose bugs belong to RETE's defect class (Section 4.3.1), as shown in Table 4.2. The programs in P28 were sampled uniformly from GitHub on 21/10/21. We exclude two samples namely wireshark-37122-37123 and gzip-3fe0caeada-39a362ae9d from MB37 used in Trident, since we could not build them along with our ML libraries. MB35 and BG107 include a test suite; the number of tests averages 62.8 for BG107 and 91.3 for MB35.

BG107	107 bugs from the BugsInPy [64] dataset.
P28	28 Python programs with artificially added holes.
MB35	The ManyBugs [51] subset used in Trident.

Table 4.2: Evaluation corpus of bugs in RETE's defect class.

P28 does not need one as we used it to train and evaluate RETE's neural variable ranker.

Experimental Setup We split the test suite into two parts (20-80), taking care that the smaller part includes at least one failing test. The smaller part is sent to a subject APR tool for patch generation. A patch is plausible if it passes the test suite. We consider a patch correct if it is plausible, and manual inspection deems it equivalent to the patch written by the developer. All the patches generated and used in this evaluation are available in the reproduction package.

For all training and fine-tuning, we used the buggy datasets with a split of 90-10 for training and testing. All the hyperparameters are tuned by using K-fold cross-validation with $k = 5$ and grid search. No layers were frozen, since we had a large enough dataset and a set of low learning rates during the grid search. We use Adam optimiser with a weight decay fix [84].

We conducted experiments inside a Docker container on a CPU of 2.7 GHz machine running on Ubuntu 21.04 with 16GB of memory and Geforce RTX 3070M. We applied a 4-hour timeout for Python-based tools and a 2-hour timeout for C-based tools. This difference is due to the fact that Python tools run tests for validation, whereas C-based tools use a patch specification for verification, which is much faster than running tests. Therefore, Python tools require more time to execute tests. In contrast, synthesis-based techniques for C, such as Trident, Angelix, and SemFix, do not need to execute tests for each possible patch, allowing for shorter timeouts.

4.5.1 Tool Configurations and Baselines

RETE's key contribution is in the variable prioritisation, which is designed to improve patch prioritisation for program repair, and is orthogonal to existing patch generation and prioritisation techniques. To evaluate its impact in isolation, we dissect existing program repair techniques into interchangeable parts, and consider

	Validator	
$\mathscr{V}$	Validation against Tests	-
$\mathscr{T}$	Trident's patch Specification	Section 3.3.2
$\mathscr{A}$	Angelix's patch Specification	[4]
$\mathscr{S}$	SOSRepair's patch Specification	[59]
	Patch Generator	
A	Angelix's Angelic Values	[4]
S	SOSRepair's Database of Snippets	[59]
P	SPR's Transformation Schemas	[9]
C	CoCoNut's Neural Machine Translation	[12]
E	Trident's Naïve template enumeration	Chapter 3
G	GenProg's Genetic Algorithm	[45]
S	Plastic Surgery Algorithm	[47]
	Patch Prioritisation	
D	DirectFix's modification minimisation	[26]
C	CoCoNut's Neural Machine Translation	[12]
E	Trident's expression size minimisation	Chapter 3
T	Trident's side effects minimisation	Section 3.3.4
P	Prophet's Maximum Likelihood Estimation	[53]
S	Plastic Surgery Algorithm	[47]
	Variable Prioritisation	
E	Naïve variable enumeration	-
H	Heuristic discussed in Section 4.5.2	-
B	CodeBERT	[78]
G	GraphCodeBERT	[85]
D	CodeBERT fine-tuned on DU chains	Section 4.3.3
C	CodeBERT fine-tuned on CDU chains	Section 4.3.3
F	Random Forest	Section 4.5.2

Table 4.3: Program Repair Components Used in Evaluation.

their combinations with RETE's approach. This analysis differs from the existing high-level classification of APR tools [86] into heuristic-based, constraint-based, and learning-based since our goal was to abstract over irrelevant components of existing tools that would complicate the objective evaluation of the proposed technique.

We analytically divided repair techniques into three main parts: patch generation, patch checking and patch prioritisation. Patch generation explores the space of patches by enumeration [9], meta-heuristics [45], neural machine translation [12, 13], or SMT-based program synthesis [8]. Patch checking determines

Configuration	Tool	Reference
$\mathscr{V}_{\text{P}}^{\text{P}}$	Prophet	[53]
$\mathscr{T}_{\text{T}}^{\text{E}}$	Trident	Chapter 3
$\mathscr{A}_{\text{D}}^{\text{A}}$	Angelix	[4]
$\mathscr{S}^{\text{S}}$	SOSRepair	[59]
$\mathscr{V}_{\text{C}}^{\text{C}}$	CoCoNut	[78]
$\mathscr{V}^{\text{G}}$	GenProg	[45]

Table 4.4: Configurations of standard tools.

whether a candidate patch meets the correctness criteria, either via validation against a test suite [45] or verification, as by solving constraints (Section 3.3.2). Patch prioritisation ranks patches by their likely correctness, as with syntactic distance [26] or machine learning [3]. We denote such techniques as $\mathscr{X}_b^a$, where $\mathscr{X}$ is a validator, a is a patch generator, and b is a patch prioritiser. For RETE, we further split patch prioritisation into template and variable prioritisation, and combine them using the technique described in Section 4.3.4. Such configurations are denoted by the notation $\mathscr{X}_{R(b,c)}^a$, where b denotes a template prioritisation technique, and c denotes a variable prioritisation technique.

The considered components are tabulated in Table 4.3. Based on this notation, Angelix [4] is $\mathscr{A}_{\text{D}}^{\text{A}}$ since it uses its own patch specification and synthesis methods while using DirectFix [26] patch prioritisation. Angelix uses logical constraints that encode program semantics and information extracted from tests, and validates candidate patches against these constraints; we refer to this validation method as A. Trident is denoted by $\mathscr{T}_{\text{T}}^{\text{E}}$ since it uses different constraints that encode information about side effects. Finally, Prophet is $\mathscr{V}_{\text{P}}^{\text{P}}$ since it extends SPR [9] using an ML-based patch prioritiser. All techniques used as baselines are tabulated in Table 4.4.

All the techniques in Table 4.3, except for CoCoNut ($\mathscr{V}_{\text{C}}^{\text{C}}$), are implemented for C. In order to make a more objective comparison, we ported Prophet to Python and trained it on Python programs in the configuration $\mathscr{V}_{\text{R(P, C)}}^{\text{P}}$. Prophet orders partially/fully instantiated patches using machine learning and concretising the partially instantiated patches using SPR's condition synthesis algorithm [9]. In SPR's algorithm, when multiple possible candidate variables have the same potential value

map, we improve Prophet by supplanting it with RETE's variable prioritisation to prioritise variables to satisfy the condition. $\mathscr{V}^{\mathrm{P}}_{\mathrm{R(P,\,C)}}$ differs from $\mathscr{V}^{\mathrm{P}}_{\mathrm{P}}$ in cases where multiple variables satisfy the conditions, $\mathscr{V}^{\mathrm{P}}_{\mathrm{R(P,\,C)}}$ uses its variable prioritiser (*i.e.* C) to order these unordered candidate variables. Section 7.2 discusses why we consider these baselines. We restrict the defect class to inserting or modifying a single atomic statement for all tools that we ran in the evaluation. This does not cover instances such as inserting for/while block, sequence of atomic statements, method declaration, *etc*. We used this constraint to make the patch generation more feasible by reducing the search space.

4.5.2 CDU Chains Contain Strong Signal

To demonstrate the variable usage signal CDU chains contain, we compare token predictors against CodeBert fine-tuned on CDU chains on our variable prediction task.

We use four variable prioritisation baselines — H, F, B, and G — introduced in Table 4.3. The first, H, ranks candidate variables based on their frequency in a program. The second, F, is the random forest implemented using explicit feature engineering; outperforming this technique means that our neural approach is outperforming manual feature engineering. More information on the features used by the random forest available in our reproduction package. These two baselines frame the results; we use the two neural baselines — CodeBert B and GraphCodeBert G — without fine-tuning to demonstrate variable signal in CDU chains. We did not fine-tune GraphCodeBert with CDU chains because it takes both source code and that code's dataflow as input. The difference between the performance of CodeBert and GraphCodeBert in Table 4.5 do not appear to warrant the engineering required to extract the dataflow GraphCodeBert needs. We note that CDU chains, by construction, implicitly contain dataflow information, which a neural network trained using them may learn and exploit.

Internally, all of these approaches rank candidate variables, so it is natural to compare them using standard rank measures. Unfortunately, none works well in our setting. We do not use mean average precision (MAP) or normalised discounted

gain (NDCG) because we only care about the rank of the first plausible patch, not their density in a prefix of the complete ranking. We do not use Mean reciprocal rank (MRR) because it does not account for search space reduction. We rank the candidate variables *w.r.t.* to the number of variables in-scope, which differs for different samples. MRR does not distinguish cases such as ranking 6^{th} out of 6 vs. 6^{th} out of 1000 variables in-scope.

Instead, we introduce a new measure, which we call the *fraction searched* measure. This measure returns the length of the prefix of ranked variables that must be checked over the total number of candidates, *i.e.* the fraction of the variable search space we had to check. For example, assume we have a search space of 100 variables and, under a particular ranker, the 16^{th} variable is the first that can fill a hole. In this case, the ranker reduced the search space to 16%. Formally, let $p \in \mathscr{L}$ be a program and $p_\square \in \mathscr{L}_\square$ be p with some of its variables replaced with holes. Let $r(\square_i, p_\square)$ be the rank that the variable ranker r assigns to the ground truth variable in p that fills $\square_i$ in $p_\square$. The r's fraction searched is

$$F(r, p, p_\square) = \frac{1}{\left|h(p_\square)\right|} \sum_{\square_i \in h(p_\square)} \frac{r(\square_i, p_\square)}{\left|var(\square_i, p_\square)\right|}$$

where $h(p_\square)$ counts the holes in $p_\square$ and $var(\square_i, p_\square) \subseteq V$ denotes the set of all the variables in-scope $\square_i$ in $p_\square$.

To compute $var(\square, p_\square)$, we consider only variables whose type matches the hole's type. For Python, this does not help much since everything is an object and many objects seamlessly slot into many expressions because of default functions, like `__bool__()`. Hence, we restrict rankings to a fixed vocabulary of variables that varies by bug. We construct this vocabulary by limiting the accessors ("`.`") to 2 to leverage the Law of Demeter [87] and avoid variable explosion.

Table 4.5 shows that CodeBERT [78], fine-tuned with CDU Chains, (Column C) outperforms the baselines. In particular, it outperforms the neural baselines — vanilla CodeBERT (*B*) and GraphCodeBert (*G*) — by an order of magnitude. We observe that CodeBERT, when fine-tuned on CDU chains, (Column C), performs better when

Table 4.5: The performance of variable prioritisation techniques using the fraction searched measure. The best configuration C, CodeBert fine-tuned with CDU chains, is bolded.

Dataset	samples	H	F	B	G	D	**C**
Python	802k	0.476	0.039	0.232	0.204	0.18	**0.033**
C	652k	0.416	0.032	0.260	0.252	0.07	**0.028**

```
-  if (tif->tif_rawcc > 0 && tif->tif_rawcc != orig_rawcc
+  if ((tif->tif_rawcc > 0)
       && (tif->tif_flags & TIFF_BEENWRITING) != 0
       && !TIFFFlushData1(tif))
     {
```

Figure 4.4: Dev. patch for libtiff-2007-11-02-371336d-s865f7b2.

compared to its counterpart fine-tuned on DU chains (Column D). Using this, we can conclude that extending DU chains with conditions helps improve performance. As expected, we observe that feature engineering allows random forest (Column F) to outperform vanilla CodeBert and GraphCodeBert, which are not benefiting from CDU chains. These results are strong evidence that CDU chains are valuable source of signal for the variable prediction task. We find:

> CDU chains contain strong signal: CodeBert fine-tuned on CDU chains (Column C in Table 4.5) outperforms the neural baselines by an order of magnitude; its performance edge over random forest (F) means fewer expensive checks of candidate patches.

Although CDU chains permit CodeBert to outperform our random forest baseline, its inference is expensive. It takes 0.65s on average to respond to queries, compared to *ca.* 0.004s for our random forest.

4.5.3 Effectiveness of RETE's Prioritisation

To understand the effectiveness of RETE's patch prioritisation, we compare component combinations for Python and then C. Below, PSH refers to the plastic surgery hypothesis and VP to variable prioritiser. For Python, we use BG107 and these component combinations:

Table 4.6: On BG107, variable and template prioritisation perform better. The best configuration is bolded.

Dataset	Samples	Correct				Plausible			
		$\mathscr{V}^{\text{E}}_{\text{R(E, F)}}$	$\mathscr{V}^{\text{S}}_{\text{R(S, H)}}$	$\mathscr{V}^{\text{S}}_{\text{R(S, F)}}$	$\mathscr{V}^{\mathbf{S}}_{\mathbf{R(S, C)}}$	$\mathscr{V}^{\text{E}}_{\text{R(E, F)}}$	$\mathscr{V}^{\text{S}}_{\text{R(S, H)}}$	$\mathscr{V}^{\text{S}}_{\text{R(S, F)}}$	$\mathscr{V}^{\mathbf{S}}_{\mathbf{R(S, C)}}$
Black	4	0	0	2	2	0	3	3	3
Fastapi	3	0	0	0	0	0	0	0	0
Httpie	3	1	1	2	2	1	2	3	3
Keras	11	0	1	2	2	0	1	7	7
Sanic	1	0	0	0	0	0	0	0	0
Y-DL	3	0	0	1	1	0	0	1	1
Spacy	2	0	0	1	1	0	0	1	1
Tqdm	4	1	1	2	2	1	1	3	3
PySnooper	1	0	1	1	1	0	1	2	2
Tornado	6	0	0	1	2	0	0	2	2
Matplotlib	8	0	2	2	2	0	2	3	3
Luigi	15	1	5	7	7	1	7	7	7
Scrapy	10	0	2	2	2	0	2	2	2
Pandas	36	0	2	4	5	0	2	4	5
Overall	107	3	15	27	**29**	3	21	38	**39**

Table 4.7: Average patch ranking for MB35: variable prioritisation indeed helps since $\mathscr{T}^{\text{S}}_{\text{R(S, C)}}$ and $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$ outperform other tools by a large margin. Ranks that cannot be assessed are marked with "-".

Bug	$\mathscr{T}^{\text{E}}_{\text{T}}$	$\mathscr{T}^{\text{E}}_{\text{R(E, F)}}$	$\mathscr{T}^{\text{S}}_{\text{R(S, H)}}$	$\mathscr{T}^{\mathbf{S}}_{\mathbf{R(S, F)}}$	$\mathscr{T}^{\mathbf{S}}_{\mathbf{R(S, C)}}$
gmp-a1d3d-f17cb	165350	2563	87339	933	843
libtiff-09e82-f2d98	13313	311	14241	552	513
libtiff-764db-2e42d	90471	7655	724	2	7
libtiff-a72cf-0a36d	$\approx 10^{12}$	-	52428627	15924	15842
libtiff-37133-865f7	$\approx 10^{18}$	-	40842	8786	8566
php-70075-5a8c9	87	51	619	890	782
php-e65d3-1d984	90471	36840	110436	5072	5439
php-63673-2adf5	61	7	52	7	3
Average	$\approx 2 \times 10^{17}$	-	6585360	**4021**	**3999**

$\mathscr{V}^{\text{S}}_{\text{R(S, C)}}$ PSH with CDU chain VP

$\mathscr{V}^{\text{S}}_{\text{R(S, F)}}$ PSH with Random Forest VP

$\mathscr{V}^{\text{S}}_{\text{R(S, H)}}$ PSH with Heuristic VP

$\mathscr{V}^{\text{E}}_{\text{R(E, F)}}$ Naïve enumeration with Random Forest VP

Table 4.6 shows that $\mathscr{V}^{\text{S}}_{\text{R(S, C)}}$ significantly outperforms all the other combinations. Combining PSH and RETE's VP does indeed improve patch synthesis since $\mathscr{V}^{\text{S}}_{\text{R(S, C)}}$ and $\mathscr{V}^{\text{S}}_{\text{R(S, F)}}$ both significantly outperform $\mathscr{V}^{\text{S}}_{\text{R(S, H)}}$ and $\mathscr{V}^{\text{E}}_{\text{R(E, F)}}$. $\mathscr{V}^{\text{E}}_{\text{R(E, F)}}$ could fix only three bugs, whereas $\mathscr{V}^{\text{S}}_{\text{R(S, F)}}$ fixes 27 and $\mathscr{V}^{\text{S}}_{\text{R(S, H)}}$ fixes 15. This is because most patches are

Table 4.8: This table compares Prophet and CoCoNut against RETE's best variant ($\mathcal{V}^{S}_{R(S,C)}$) and Prophet enhanced with RETE ($\mathcal{V}^{P}_{R(P,C)}$). For correct and plausible patches, RETE outperforms the other baselines by generating 7 and 13 additional correct patches against Prophet and CoCoNut, respectively. All the RETE variants are bolded.

Dataset	Samples	Correct				Plausible			
		$\mathcal{V}^{P}_{P}$	$\mathcal{V}^{P}_{R(P,C)}$	$\mathcal{V}^{C}_{C}$	$\mathcal{V}^{S}_{R(S,C)}$	$\mathcal{V}^{P}_{P}$	$\mathcal{V}^{P}_{R(P,C)}$	$\mathcal{V}^{C}_{C}$	$\mathcal{V}^{S}_{R(S,C)}$
Black	4	1	2	1	2	3	3	3	3
FastApi	3	0	0	0	0	0	0	0	0
Httpie	3	1	2	1	2	3	3	2	3
Keras	11	2	2	1	2	4	4	3	7
Sanic	1	0	0	0	0	0	0	0	0
Y-DL	3	0	0	0	1	0	0	0	1
Spacy	2	1	1	0	1	1	1	1	1
Tqdm	4	1	2	1	2	2	2	1	3
PySnooper	1	1	1	1	1	1	1	1	2
Tornado	6	0	0	0	2	0	0	0	2
Matplotlib	8	2	2	2	2	2	2	2	3
Luigi	15	8	8	5	7	8	8	8	7
Scrapy	10	2	2	1	2	2	2	3	2
Pandas	36	3	3	3	5	5	5	6	5
Overall	107	22	**25**	16	**29**	31	**31**	30	**39**

Table 4.9: The number of patches that require non-local variables. A variable is considered non-local if it is not directly used in the method. All RETE configurations are bolded. Using RETE with variable prioritisation helps generate non-local variables. The results show that CoCoNut ($\mathcal{V}^{C}_{C}$) and Prophet ($\mathcal{V}^{P}_{P}$) could not generate patches with long ranged dependencies on their own.

	$\mathcal{V}^{C}_{C}$	$\mathcal{V}^{P}_{P}$	$\mathcal{V}^{P}_{R(P,C)}$	$\mathcal{V}^{S}_{R(S,C)}$
Correct Patches	16	22	25	29
Correct Patches with non-local Variables	0	0	**1**	**5**

closer to pre-existing code located somewhere in the buggy program's code [47]; hence, when $\mathcal{V}^{E}_{R(E,F)}$ tries to construct a patch in such cases, it faces the harder task of doing so from the ground up without guidance. $\mathcal{V}^{S}_{R(S,F)}$ fixes more bugs than $\mathcal{V}^{S}_{R(S,H)}$, since it enumerates patches more intelligently (Section 4.3.3). $\mathcal{V}^{S}_{R(S,C)}$ fix two more correct bugs when compared against $\mathcal{V}^{S}_{R(S,F)}$ since CDUs identify relevant variables better than our random forest baseline.

For C, we utilised the eight bugs in intersection of MB35 and the subset of RETE 's defect class on which its best configuration produces a fix. We leverage Trident's multi-path sepcification inference (Section 3.3.2) as a validator: C compiles

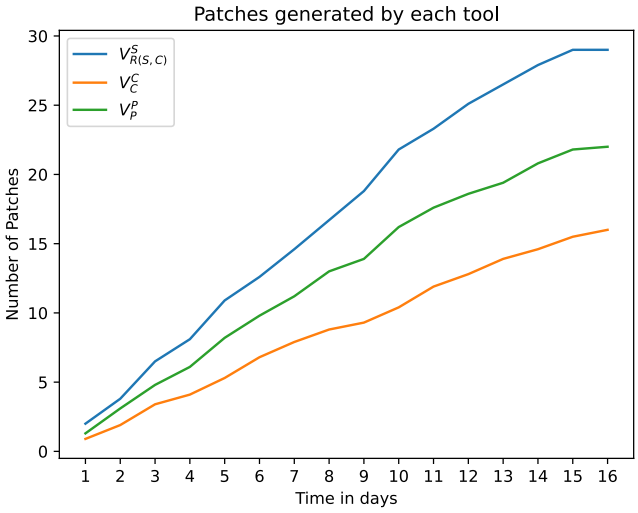


Figure 4.5: Patches generated per unit time by RETE ($\mathscr{V}^{\text{S}}_{\text{R(S, C)}}$), Prophet ($\mathscr{V}^{\text{P}}_{\text{P}}$) and CoCoNut ($\mathscr{V}^{\text{C}}_{\text{C}}$).

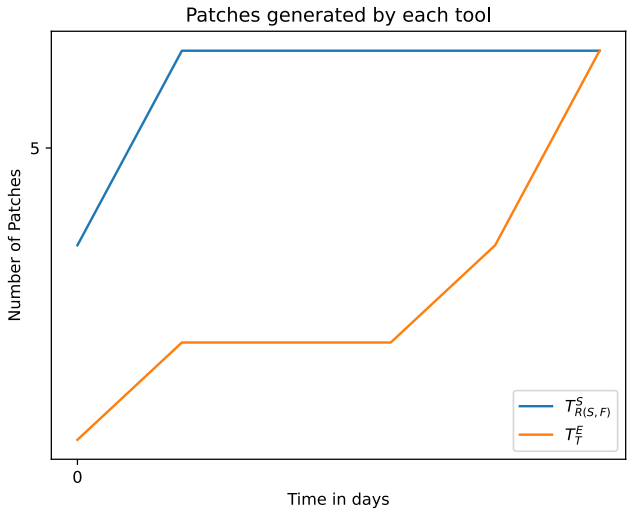


Figure 4.6: The number of patches generated by $\mathscr{T}^{\text{E}}_{\text{T}}$ and $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$ per unit time: $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$ generates patches faster than $\mathscr{T}^{\text{E}}_{\text{T}}$.

slowly, and Trident's efficient patch specification inference obviates many compilations. We compare these five component combinations: $\mathscr{T}^{\text{E}}_{\text{T}}$ (vanilla Trident), $\mathscr{T}^{\text{E}}_{\text{R(E, F)}}$, $\mathscr{T}^{\text{S}}_{\text{R(S, H)}}$, $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$ and $\mathscr{T}^{\text{S}}_{\text{R(S, C)}}$. Table 5.6 shows the results. In some cases, the ranks of patches from $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$ and $\mathscr{T}^{\text{S}}_{\text{R(S, C)}}$ are extremely low compared to those $\mathscr{T}^{\text{E}}_{\text{T}}$'s patches. This is because $\mathscr{T}^{\text{E}}_{\text{T}}$ constructs statements from ground up, hence bugs such as `libtiff-a72cf60-0a36d7f` and `libtiff-371336d-865f7b2` (Figure 4.4) cannot be synthesized by $\mathscr{T}^{\text{E}}_{\text{T}}$ as they require constructing a fresh expression

Table 4.10: Plausible and correct patches for bugs in MB35 (C dataset). RETE integrated with Trident, the column labelled $\mathscr{T}^{S}_{R(S,\,F)}$, generates 18 plausible patches out of 35, of which 8 are correct. The next largest total, GenProg, the column labelled $\mathscr{V}^{G}$, generates 19, of which only two are correct. No other tool configuration generates more correct patches than $\mathscr{T}^{S}_{R(S,\,F)}$.

Bug	kLoC	Total	Plausible						Correct					
			$\mathscr{T}^{S}_{\mathbf{R(S,\,F)}}$	$\mathscr{T}^{E}_{T}$	$\mathscr{A}^{A}_{D}$	$\mathscr{V}^{P}_{P}$	$\mathscr{S}^{S}$	$\mathscr{V}^{G}$	$\mathscr{T}^{S}_{\mathbf{R(S,\,F)}}$	$\mathscr{T}^{E}_{T}$	$\mathscr{A}^{A}_{D}$	$\mathscr{V}^{P}_{P}$	$\mathscr{S}^{S}$	$\mathscr{V}^{G}$
gmp	145	2	1	1	2	1	0	1	0	0	0	0	0	0
gzip	491	3	3	3	3	2	0	1	1	1	1	1	0	0
libtiff	77	10	8	5	5	5	8	7	4	3	2	1	2	2
php	1,099	19	6	6	6	9	4	9	3	3	3	4	3	0
wireshark	2,814	1	1	1	1	1	1	1	0	0	0	0	1	0
Overall		35	**18**	15	17	18	13	19	**8**	6	6	6	6	2

with 8-13 different variables and operators. In such cases, we resorted to estimating $\mathscr{T}^{E}_{T}$'s rank on them by assuming that candidate variables were uniformly ordered. Unfortunately, we could not estimate $\mathscr{T}^{E}_{R(E,\,F)}$'s performance the same way, since it uses F as its VP. On average, the search space is reduced by a few orders of magnitude on these seven bugs. The rankings of configurations $\mathscr{T}^{S}_{R(S,\,H)}$ and $\mathscr{T}^{S}_{R(S,\,C)}$ affirm that both RETE's template and variable prioritisers play a pivotal role in synthesising patches.

> On both our Python and C datasets, enhancing other techniques with RETE's template and variable prioritisers improves their performance.

4.5.4 RETE's Performance

We now show that RETE advances the state of the art in 1) time to generate patches and 2) number of correct patches. We close by showing how it speeds Trident, the state of the art C APR tool for handling side effects.

Figure 4.5 shows how many patches RETE ($\mathscr{V}^{S}_{R(S,\,C)}$), Prophet ($\mathscr{V}^{P}_{P}$) and CoCoNut ($\mathscr{V}^{C}_{C}$) produce per unit time. All 107 programs from the dataset are uniformly ordered and then each tool's execution time to patch each bug, with a timeout of four hours, is summed, upto the time budget. For each budget, we repeat this process 10 times with 10 different orders and average the results. At all budgets, RETE ($\mathscr{V}^{S}_{R(S,\,C)}$) outperforms Prophet and CoCoNut.

Table 4.8 compares RETE against Prophet [3] and CoCoNut [78] in terms of

the number of correct and plausible patches each generates. All RETE variants are bolded in the table. Prophet ($\mathscr{V}^{P}_{P}$) generates 22 and CoCoNut ($\mathscr{V}^{C}_{C}$) 16 correct patches. RETE's best configuration ($\mathscr{V}^{S}_{R(S,\,C)}$) generates 29 correct patches. On Python, RETE improves the state of the art by 31% vs. Prophet and 59% vs. CoCoNut. Prophet, enhanced with RETE's variable prioritisation ($\mathscr{V}^{P}_{R(P,\,C)}$), fixes three more bugs than vanilla Prophet [3]. Interestingly, we observe each of these two tools exclusively fixes some patches, like patches in `Luigi` and `Pandas`. Although $\mathscr{V}^{P}_{R(P,\,C)}$ generates more correct patches than Prophet, they generate the same number of total patches. This may indicate that Prophet enhanced with Rete's variable prioritisation reduces the overall overfitting of the generated patches.

A variable is non-local if it is not used in the buggy method. For example, if the variable `var` is directly used somewhere in the method, but `var.a` is not, `var` is local and `var.a` is non-local. Under this definition, most in-scope variables in Python are non-local. Table 4.9 shows that integrating variable prioritisation helps synthesise patches that require non-local variables: $\mathscr{V}^{S}_{R(S,\,C)}$ fixes five bugs that require non-local variables, whereas the other tools struggle to synthesise correct patches that require them.

> $\mathscr{V}^{S}_{R(S,\,C)}$ fixes five bugs that require non-local variables, whereas other tools struggle to synthesise correct patches that require non-local variables.

Table 4.10 compares the performance of RETE-enhanced Trident ($\mathscr{T}^{S}_{R(S,\,F)}$) against the state of the art. We did not include $\mathscr{T}^{S}_{R(S,\,C)}$ for C, despite the fact that it generates better rankings, because Trident's validation step is much faster on average than querying CodeBert (0.28 vs. 0.65 seconds). A faster GPU than ours would probably alleviate this issue.

Although $\mathscr{T}^{S}_{R(S,\,F)}$ does not synthesise new patches relative to the state of the art, it does synthesise more correct patches. Angelix also generated the two correct patches that $\mathscr{T}^{S}_{R(S,\,F)}$ generated beyond those generated by vanilla Trident. This is due to patch construction. Angelix ($\mathscr{A}^{A}_{D}$) constructs patches by minimally modifying the existing expressions using SMT queries, while RETE's PSH template constructor

tries to find patches close to a set of existing statements. One such patch is Figure 4.4. Here, Angelix and $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$ both successfully generate this patch, since it is easy to modify the existing expression to reach the patch by simply dropping the expression `tif->tif_rawcc != orig_rawcc`. Vanilla Trident, in contrast, does not since the patch requires an expression with 11 components, which is infeasible due to combinatoric explosion. To understand the importance of RETE's variable prioritisation more clearly, we ran an experiment by varying the time limits on these seven bugs that $\mathscr{T}^{\text{E}}_{\text{T}}$ synthesises to compare it against $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$. Figure 4.6 shows that $\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$ synthesises patches much faster than $\mathscr{T}^{\text{E}}_{\text{T}}$.

RETE-enhanced Trident ($\mathscr{T}^{\text{S}}_{\text{R(S, F)}}$) repairs bugs faster than vanilla Trident ($\mathscr{T}^{\text{E}}_{\text{T}}$) while fixing two more bugs.

Chapter 5

Precise Data-Driven Approximation for Program Analysis

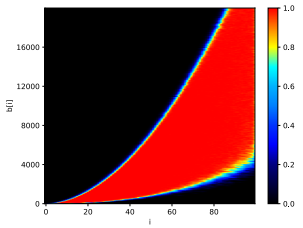
5.1 Introduction

Program analysis checks if a given program satisfies certain correctness properties. Due to the undecidability, program analysers approximate program behaviour, which results in imprecision. Techniques like abstract interpretation (Section 2.2.2) overapproximate program behaviour, causing false positives (false alarms), and techniques like symbolic execution (Section 2.2.3) underapproximate program behaviour [88, 89], which leads to false negatives (missed violations). Abstract interpretation resort to coarse abstractions of program states to make their analysis decidable, at the cost of precision. On the other hand, symbolic execution is imprecise because it explores a finite number of execution paths and thus may miss important edge cases.

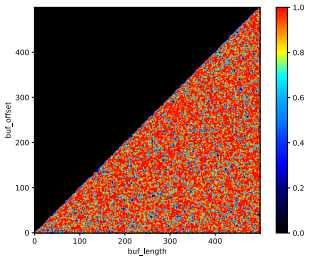
To improve the precision and recall of program analysis, we propose an alternative to underapproximate and overapproximate reasoning. Our approach leverages execution samples to probabilistically approximate program states via a novel data structure called *program state probability* (PSP). For each state, PSP estimates the probability of existence of a program input that reaches that particular state during execution. We compute PSP by constructing an estimator, such as random forest, from data obtained through fuzzing.

```
int N, off;
scanf("%d", &N);
if (N <= 0) return 1;
ll *a = malloc(sizeof(ll) * N);
ll *b = malloc(sizeof(ll) * N);
for (int i = 0; i < N; i++) {
    scanf("%d", &off);
    if (off < 0 || off > 6)
        return 1;
    int p = i - 1;
    a[i] = (i == 0)?6 : a[p] + off;
    b[i] = (i == 0)?0 : b[p] + a[p];
    assert(b[i] <= 3*i*i + 3*i);
}
return 1;
```

(a) A program from SV-benchmarks for which Clam produces a false positive (`ll` denotes **`long long`**).



(b) A heatmap showing the probability of reaching programs states *w.r.t.* the values of `b[i]` and `i` (cooler means less likely).



(c) A heatmap computed when analysing an assertion in OpenSSL, for which Clam generates a false positive.

Figure 5.1: An example program, a PSP heatmap computed for this program, and a heatmap for analysis of OpenSSL.

This technique's fundamental premise is that significant variation exists in the likelihood of reaching various states. Therefore, we can considerably enhance analysis precision at the cost of a small probability of unsoundness or incompleteness. First, PSP enables us to disregard low-probability states deemed feasible by over-approximation, thus reducing false positives. Second, PSP enables us to consider high-probability states deemed infeasible by underapproximation, thus reducing false negatives. The level at which low-probability states are ignored and high-probability states are recognised is controlled by a user-defined threshold, which functions as a control knob for the analysis trade-off. We demonstrate the utility of PSP by applying it to tackle fundamental challenges faced by two widely-used analysis techniques — abstract interpretation and symbolic execution. PSP reduces abstract interpretation's false positives and increases the number of violations detected by symbolic execution. Apart from that, we applied PSP to test-driven program repair to alleviate test-overfitting by prioritising correct patches.

Abstract interpretation approximates program states using abstract domains to make analysis scalable, which often leads to false positives, causing developers to ignore legitimate warnings [90]. Minimising false positives is crucial to ensure the usefulness of abstract interpretation. To achieve that, we propose augmenting abstract domains with program state probabilities constructed using values from

fuzzing. PSP allows us to disregard low-probability states deemed reachable by the abstraction. Although this creates a low probability of unsoundness, we demonstrate that PSP effectively reduces false positives, which is important for the practical use of abstract interpretation as a bug-catching tool.

Symbolic execution executes a program with symbolic inputs and creates path conditions for the explored paths, which are solved using an SMT solver to identify feasible paths and potential bugs [91]. However, it suffers from a scalability bottleneck due to the high computational cost of constraint solving. We applied PSP to reduce the solving cost by identifying probable states and guessing which path conditions are satisfiable. Specifically, we prioritise exploring paths that are more likely to be feasible and skip solving path conditions whose probability of being satisfiable is above a user-defined threshold. As a result, PSP facilitates deeper code exploration and thus enables symbolic execution to find bugs faster.

Program repair tools face the challenge of searching for a correct patch in a search space that contains many irrelevant or incorrect patches. One way to address this challenge is to prioritise patches in a way that increases the likelihood of finding a correct patch. We formulate a new hypothesis that the set of values most variables in a program can take is invariant with respect to small edits to the program. Relying on this hypothesis, we prioritise patches that make minimal changes to the range of all values program variables can take during execution. PSP enables us to efficiently implement this prioritisation strategy by probabilistically quantifying unchanged bindings of variable to values.

We implemented PSP using AFL fuzzer [92] for C programs, and using Fuzzer Harvey [93] for smart contracts. For abstract interpretation, we integrated PSP with Clam analyser [94] for C programs. An evaluation on six programs from Magma benchmark [95] revealed that PSP enhances the precision of Clam by improving its F1-score by 74.3%. For symbolic execution, we compared our PSP-based search strategy with the state-of-the-art search strategy using pending constraints [96] and abstract symbolic execution (ASE) [97]. To compare with the pending constraints, we implemented our strategy in Mythril [98] symbolic executor for smart contracts, and

to compare with ASE, we implemented PSP inside the ASE tool. Our experiments demonstrated that PSP increases the number of found bugs by 4.1% compared to the pending constraints, and reduces the number of solver calls by 77 times compared to ASE. For program repair, we integrated PSP with the state-of-the-art patch prioritisation strategy of Rete [99]. For bugs from ManyBugs benchmark [51], PSP decreased the average rank of correct patches by 26%.

In this chapter, we make the following contributions:

- Introduce program state probability (PSP), which probabilistically approximates program states.
- Apply PSP to reduce false positives during static analysis.
- Propose a search strategy relying on PSP to guide symbolic execution to find more bugs.
- Propose patch prioritisation strategy using PSP to effectively prioritise correct patches for program repair.

PSP's implementation, and the scripts and data used to evaluate it can be found in the accompanying package `https://zenodo.org/record/7902213`

5.2 Overview

In this section, we discuss the general intuition behind program state probability, and illustrate one of its applications: reducing false positives of static analysis.

Figure 5.1a depicts a simplified code fragment from SV-benchmarks, which serves to evaluate the accuracy of program analysis tools. The program comprises a sequence of memory allocations and assignments, and includes a loop and an assertion. The assertion checks whether `b[i] <= 3*i*i + 3*i` holds for all values of i between 0 and $N-1$. Clam [94], an LLVM-based abstract interpreter for C programs, can only imprecisely analyse the code and yields intervals in the range $b = [8, \infty]$, which is inaccurate and leads to a false positive. This limitation arises because Clam does not precisely capture program dependencies because of the overapproximation of program states.

To overcome this issue, we augment Clam's approximation with a probability distribution constructed from fuzzing data. PSP is visualised in Figure 5.1b that shows a heatmap obtained from fuzzing data to construct a probability map for various program states with respect to the variables `b[i]` and `i`. The cooler colours represent a lower probability that the program can generate such a program state. The heatmap reveals a more precise range of values for b, making it more accurate than Clam while capturing the dependencies between `b[i]` and `i`.

The upper curve in the heatmap corresponds to the function `3*i*i + 3*i`, which satisfies the assertion. The lower bound provided by the heatmap is close to the actual lower bound, which is $6i$. At large values of i, it does not precisely match because we did not provide enough seed inputs for the fuzzer. However, it is sufficient to conclude that the probability that there exists an input that leads to the assertion violation is 0.1826, which is below our default threshold of 0.8. Hence, PSP helps to eliminate Clam's false positive.

The heatmap in Figure 5.1c is obtained by analysing a larger program, OpenSSL, which contains the assertion `assert(ctx->buf_len >= ctx->buf_offset)`. The heatmap shows the distribution of values for the variables `ctx->buf_len` and `ctx->buf_offset`. We can see that the heatmap's heat extends up to the location of the assertion, indicating that the condition holds with a high probability based on the output of PSP. In contrast, Clam does not effectively capture the relationship between these variables, which leads to a false positive.

5.3 Program State Probability

This work rests on the idea of using the probability that a program can reach a given program state to precisely approximate program behaviour. Here, we formalise program state probability, detail how to calculate it, and apply it to estimate the likelihood of a program condition being satisfiable.

We use the following terminology. Let I be the set of all possible inputs a program f can take. Let V be f's variables and X be the values a variable can take. Let Σ^f be the set of all concrete states the program f can potentially take.

Each state $\sigma : V \to X \in \Sigma$ maps variables to values. We assume that V contains the program counter, which binds each state to the location in the program at which it is computed. Consider the set of abstract states (sets of concrete states) computed during an abstract interpretation of the program f. Let A_f be the union of all the abstract states computed during the abstract interpretation of f, so $A_f \subseteq \Sigma^f$.

5.3.1 Estimating Program State Probability

Let f be a program, let $\sigma \in \Sigma^f$ be an arbitrary program state of f, and let $f(i) = \sigma_0, \sigma_1, \ldots, \sigma_n = \overline{\sigma}$ denote the execution of f instrumented to output its entire state trajectory. This indicator function defines the set of states f can generate:

$$\mathbb{I}_{\Sigma^f}(\sigma) = \begin{cases} 1 & \text{if } \exists i \in I, \sigma \in \{\sigma_j \mid f(i) = \overline{\sigma} \wedge j \in [0..|\overline{\sigma}|]\} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

This indicator function is not computable, so we define program state probability to approximate it.

Definition 17 (Program State Probability (PSP))**.** *For the program f, the* program state probability *of the program state $\sigma \in \Sigma^f$ is the probability that f can generate σ. Formally, it is $P(\mathbb{I}_{\Sigma^f}(\sigma) = 1)$.*

To estimate PSP, we use the information generated from fuzzing. We represent the event of running a fuzzer as F and the fuzzing data using the multiset *Fuzz* whose element multiplicity represents the number of times the program state σ was encountered n times during the fuzzing campaign.

We use the *Fuzz* produced by fuzzing campaign F to estimate PSP as follows:

$$P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F) = \begin{cases} 1 & \sigma \in Fuzz \\ 0 & \sigma \notin A_f \\ r(\sigma, Fuzz) & \text{otherwise} \end{cases} \tag{5.2}$$

where r is a probability distribution, which defaults to uniform, over the states unseen during the fuzzing campaign F.

When the fuzzing campaign encountered σ, we can say with certainty that $P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F) = 1$. When $\sigma \notin A_f$, we can say with certainty that $P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F) = 0$, as abstract domain overapproximates the set of possible program states. Otherwise, we use the function r to estimate, or "radiate", probability f can generate σ from the observations in *Fuzz*. This function can be instantiated in various ways, as discussed in Section 5.3.2. Where clear from context, we use $P(\mathbb{I}_{\Sigma^f}(\sigma))$ to denote $P(\mathbb{I}_{\Sigma^f}(\sigma) \mid F)$.

We compute the probability of a state's feasibility, such that if the state is discovered during the fuzzing campaign ($\sigma \in \mathit{Fuzz}$), its probability is set to 1, indicating that the state is feasible. The number of times the state is encountered during the fuzzing campaign does not affect this measure. Therefore, this approach is resistant to changes in the fuzzing settings, although it is not completely immune to them.

5.3.2 Estimating Unseen States

Empirically, we observe that variable bindings often obey rules and exhibit patterns, like monotonically increasing at a fixed stride, only taking on odd (or even) numbers, or oscillating among a few error codes. We leverage this insight to estimate $r(\sigma, \mathit{Fuzz})$. A program generates very few of the possible program states. We first check whether σ's neighbourhood has enough data to contend with this sparsity. If it does, we look for patterns in it. If we discern a pattern, we increase the probability of unseen values that obey it. We use smoothing to reserve probability weight for the rest of the values.

We use supervised learning to find binding patterns. Let $N\colon \Sigma \times \mathbb{N} \to 2^{\Sigma}$ be σ's k-neighbors:

$$N(\sigma, k) = \{\sigma' \in \Sigma \mid d(\sigma, \sigma') \leq k\}$$

where d is a distance function over program states that share almost all their variables. We assume d ignores unshared variables. In N, k is the neighbourhood radius and limits the neighbourhood size. We apply N to the bindings recorded in *Fuzz*, the output of the fuzzing campaign F. Suppose the population of a state's neighbourhood

exceeds the minimum sample threshold u. In that case, we employ a supervised model M_s to estimate $r(\sigma, Fuzz)$ and predict states in a given observed state's neighbourhood that were unseen during fuzzing; otherwise, we resort to the unsupervised statistical method $M_{\bar{s}}$:

$$r(\sigma, Fuzz) = \begin{cases} M_s(O) & |N(\sigma,k)| \geq u \\ M_{\bar{s}}(Fuzz) & \text{otherwise} \end{cases} \tag{5.3}$$

where $O = \{\sigma' - \sigma \mid \sigma' \in N(\sigma,k)\}$.

We use O to offset each state in $N(\sigma,k)$ by σ before giving it to the model. This way, the model only needs to predict the probability of a particular pattern of variable values, rather than the exact values themselves. For example, if a variable only takes on odd or even values, the model can learn to predict the probability of the variable being odd or even without being affected by the absolute values of the variable.

To estimate $M_{\bar{s}}(Fuzz)$, we can subject the empirical distribution that the multiset $Fuzz$ defines to various classical smoothing techniques such as kernel density estimation. Alternately, we could use heat diffusion to distribute probabilities across neighbouring points:

$$\frac{\partial P(\mathbb{I}(\sigma))}{\partial t} = \nabla^2 P(\mathbb{I}(\sigma)) + Q(\sigma, t) \tag{5.4}$$

The variable t represents the time during which the system evolves. The function $Q(\sigma,t)$ supplies or removes heat from the system, ensuring that the heat sources and sinks (i.e., states encountered during fuzzing and states not in the abstract domain) maintain their probability. Specifically, $\forall t \in [0,\infty],\ \sigma \in Fuzz \Rightarrow P(\mathbb{I}(\sigma \mid F)) = 1$ and $\sigma \notin A \Rightarrow P(\mathbb{I}(\sigma \mid F)) = 0$ We numerically solve Equation (5.4) using the Forward Time Centered Space method [100]. We evaluated these different approaches in Section 5.6.5; our implementation uses the heat diffusion method because it worked best as shown in Section 5.5.2.

5.3.3 Satisfiability Under PSP

Consider an arbitrary condition C in the program f containing variables $v_1 \dots v_n$. We denote the probability that there exists an input that satisfies this condition as $P(\mathbb{I}_C^f)$ where:

$$\mathbb{I}_C^f = \begin{cases} 1 & \text{if } \exists \sigma \in \Sigma^f, \mathbb{I}_\Sigma^f(\sigma) = 1 \wedge \langle C, \sigma \rangle \Downarrow 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

where $\Downarrow$ evaluates C in the program state σ.

We compute the lower bound of $P(\mathbb{I}_C^f)$ by assuming the event with the maximum probability subsumes all other events. Consider the set $\Sigma_C^f = \{\sigma \in \Sigma^f, \langle \sigma, c \rangle \Downarrow 1\}$. Using this assumption and utilising Σ_C^f gives us the following expression:

$$P(\mathbb{I}_C^f) \geq \max_{\sigma \in \Sigma_C^f} P(\mathbb{I}_{\Sigma^f}(\sigma)) \tag{5.6}$$

We can similarly compute the upper bound as follows:

$$P(\mathbb{I}_C^f) \leq \min\big(1, \sum_{\sigma \in \Sigma_C^f} P(\mathbb{I}_{\Sigma^f}(\sigma)\big) \tag{5.7}$$

We can compute a weak approximation of $P(\mathbb{I}_C^f)$ by assuming that the set of events $\Sigma_C^f = \{\mathbb{I}_\Sigma^f(\sigma_i) \mid \sigma_i \in \Sigma, \langle C, \sigma_i \rangle \Downarrow 1\}$ are mutually independent:

$$P(\mathbb{I}^f(C)) \approx 1 - \prod_{\sigma_i \in \Sigma_C^f} \Big(1 - P\big(\mathbb{I}_{\Sigma^f}(\sigma_i)\big)\Big) \tag{5.8}$$

5.4 PSP Applications

Here, we introduce three applications of PSP: enhancing the precision of static analysis, optimising symbolic execution, and improving the quality of automatically generated patches.

5.4.1 Static Analysis

Abstract interpretation overapproximates program states to make analysis scalable, which leads to false positives, discouraging developers from examining analysis

violations. Therefore, minimising false positives is crucial for the usability of abstract interpretation. To achieve that, we use PSP to construct $P(\mathbb{I}(\sigma))$ by relying on fuzzing data as well as the abstract states A_f computed by an abstract interpretation tool. Then, we calculate $P(\mathbb{I}^f_{\neg C})$ for each assertion C, as explained in Section 5.3.3. We only report an assertion violation if the resulting probability is greater than a user-defined threshold θ. This enables us to disregard low-probability states deemed reachable by overapproximation, thus reducing false positives. The threshold θ is set such that for $\theta = 0$, we have complete overapproximation, while for $\theta = 1$, we have complete soundness. To ensure soundness at $\theta = 1$, we restrict the estimation of probability using statistical methods to be strictly less than 1. This means that if a probability is estimated as 1, its value is set to $1 - \varepsilon$, where $\varepsilon \to 0^+$. This restriction enforces soundness at $\theta = 1$, and ensures that we are not reporting false positives due to imprecise probability estimates.

5.4.2 Symbolic Execution

Symbolic execution constructs a path condition, the conjunction of conditionals it encounters along a path. It queries an SMT solver to determine which paths are feasible by checking the satisfiability of their path conditions, which is computationally expensive. PSP can reduce this cost by identifying probable states and guessing if path conditions are satisfiable.

We utilise PSP to prioritise execution paths when symbolically executing a subject program to find bugs. The algorithm is shown in Algorithm 3. As usual, it takes the subject program and a termination condition, which could bound steps, time, memory, or computation. Algorithm 3's key novelties are two-fold: 1) it prioritises probable paths, line 10, and 2) it skips, line 18, solving probable program states, only checking whether improbable states are satisfiable (line 19). The *solve* helper function extracts the path condition from the symbolic state s and invokes an SMT solver. The $bugs(s)$ function checks whether the symbolic state s can possibly break certain criteria drawn from the semi-universal implicit specification of most programs, such as integer or buffer overflows.

Because of its new features, Algorithm 3 spends more time exploring probable

Algorithm 3: This algorithm shows symbolic execution using PSP.

```
1  Input:
2  f: The subject program
3  PSP: The PSP model built from fuzzing data
4  θ: Threshold for ignoring improbable states
5  κ: Termination criteria
6  Output:
7  B: The set of bugs discovered
8
9  S_0 := initState(f)
10 workList := PriorityQueue(PSP)
11 workList.put(S_0)
12 B := {}
13 while workList ≠ ∅ ∧ κ do
14     s := workList.next()
15     B := B ∪ bugs(s)
16     newStates := execute(s)
17     for s ∈ newStates do
18         if PSP(I^f_{s.pc}) < θ then
19             if solve(s) ≠ SAT then
20                 continue
21         workList.put(s)
22 return B
```

paths rather than solving constraints. Despite not solving probable states, Algorithm 3 always explicitly solves and stores the inputs for those bugs it finds, *i.e.*, $solve(s) \neq SAT \implies bugs(s) = \emptyset$. It may, of course, deem some UNSAT state to be probable, thus treating it as SAT and generating a false negative. To support its new features, Algorithm 3 additionally takes, as input, a PSP model trained on a fuzzing campaign F's fuzzing data and a threshold value θ that determines which program states are sufficiently improbable to ignore. Finally, as usual, it returns the set of bugs it finds during its run.

We cannot directly employ the bounds discussed in Section 5.3.3 as it is not feasible to enumerate through all possible program states without any approximation. Hence, we aim to directly estimate $P(\mathbb{I}^f_{pc})$. We represent this condition using Conjunctive Normal Form (CNF). Any path condition pc can be represented as $pc = C' \wedge C$ where C is a conjunct-free condition. We can compute the probability

as $P(\mathbb{I}^f_{pc}) = P(\mathbb{I}^f_C \mid \mathbb{I}^f_{C'}) \times P(\mathbb{I}^f_{C'})$, by effectively computing $P(\mathbb{I}^f_C \mid \mathbb{I}^f_{C'})$, we can recursively repeat this to evaluate this condition completely. This recursive process is not necessary as the path conditions are incrementally built, hence when a new condition C gets added, we will have $P(\mathbb{I}^f_{pc'}) = P(\mathbb{I}^f_C \mid \mathbb{I}^f_{pc}) \times P(\mathbb{I}^f_{pc})$, since the probability of the previous patch condition has already been computed, we are just left with having to compute $P(\mathbb{I}^f_C \mid \mathbb{I}^f_{pc})$.

To effectively evaluate $P(\mathbb{I}^f_C \mid \mathbb{I}^f_{C'})$, we employ the χ^2 test of independence offline to prune out irrelevant conditions. We perform this action by constructing a dependency graph $g : V \times V \to \{0,1\}$, which returns true if two variables are dependent. Using this graph, we can effectively eliminate conjuncts from C' that are possibly independent and drop them since for any two independent events A and B, $P(A \mid B) = P(A)$. Using this dependency graph, any conjunct in C' which does not contain any variables which belong to the same connected component of at least one variable used in C is dropped. The next step is to filter the *Fuzz* data to only include program states that pass through the points guarded by the remaining constraints. This results in a new set of data denoted as $Fuzz'$. We then use $Fuzz'$ to evaluate $P(I^f_C \mid \mathbb{I}^f_{C'})$ using Equation (5.6), which provides the most conservative bound and prioritizes precision. As this reduces our chances of giving an unfeasible path a higher probability of SAT.

5.4.3 Patch Prioritisation

Program repair tools search for a patch in a search space that includes many irrelevant or incorrect patches. Due to the sparsity of useful patches in this space, brute force enumeration is usually infeasible. So instead, patch prioritisation orders the patches so that the correct ones appear earlier during enumeration.

Since program repair tools typically generate patches, whose size is insignificant *w.r.t.* to the size of the program, we formulate the following hypothesis:

Hypothesis 1. *The set of values that most variables in a program can take is invariant* w.r.t. *the edits δ iff δ change a sufficiently small portion of the program.*

We acknowledge that counterexamples for this hypothesis exist. Nonetheless,

we observed that it often holds in practice. Relying on this hypothesis, we prioritise patches that make minimal changes to the range of all values program variables can take during execution. We speculate this prioritisation strategy can be used in conjunction with other strategies to increase the quality of automatically generated patches.

In practice, a patch δ usually involves replacing one variable with another with an indistinguishable distribution over its values, thus our focus here on probabilistically quantifying probabilistically unchanged bindings. To probabilistically check Hypothesis 1, we first define the subset of a program's state on which an arbitrary predicate holds:

$$\mathbb{I}_{\Sigma^f}(\alpha) = \begin{cases} 1 & \text{if } \{\sigma \mid \sigma \in \Sigma^f, \alpha(\sigma)\} \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

We are interested in computing the probability that the program patch δ does not change some binding when $\delta(f)$ is executed. For the fixed binding $v = x$, we leverage this indicator function over predicates to define

$$D(v,x) = \left| P(\mathbb{I}_{\Sigma^f}(\sigma(v) = x)) - P(\mathbb{I}_{\Sigma^{\delta(f)}}(\sigma(v) = x)) \right|$$

This difference captures the probability that the two program variants disagree on the binding $v = x$; when they are in perfect agreement, this difference is zero. Finally, we define the probability that there exists an input on which the program patch δ changes the binding of a variable that f and $\delta(f)$ share:

$$\mathbb{P}(f,\delta) = \prod_{v \in V \cap V_\delta} \prod_{x \in X} \big(1 - D(v,x)\big) \tag{5.10}$$

We cannot directly compute Equation (5.10), because X is often infinite. Hence, we approximate X by the values observed during the execution of a test suite T on f and $\delta(f)$. We denote these constrained sets as $X|T(f)$ and $X|T(\delta(f))$ and constrain x to their union.

Since the number of patches is large, the probability computation must be efficient. Hence we constrain v to δ_V over V, which denotes all the variables in the patch δ. This reduces the number of variables being considered. A limitation of this approximation is that when the patch does not change any form of assignment (such as modifying conditions), the newly constrained equation may not capture any changes, rendering the probability calculation unnecessary.

Below, we use b to denote the binding "$\sigma(v) = x$". We can speed Equation (5.10)'s computation by precomputing $P(\mathbb{I}_{\Sigma f}(b))$, using Equation (5.9). To construct $P(\mathbb{I}_{\Sigma^{\delta(f)}}(b))$, since we restrict our equation to $v \in V_\delta$ and $x \in X|T(f)$, we approximate $P(\mathbb{I}_{\Sigma^{\delta(f)}}(b))$ as follows

$$P(\mathbb{I}_{\Sigma^{\delta(f)}}(b)) \approx \begin{cases} 1 & x \in X|T(\delta(p)) \\ P(\mathbb{I}_{\Sigma f}(b)) & \text{otherwise} \end{cases}$$

We use Section 5.4.3 to transform Equation (5.10) as follows

$$\mathbb{P}(f,\delta) = \prod_{v \in V_\delta} \prod_{x \in X|T(\delta(f))} \big(1 - |1 - P(\mathbb{I}_{\Sigma f}(b)|\big) \tag{5.11}$$

We can replace $P(\mathbb{I}_{\Sigma^{\delta(f)}}(b))$ with 1, from Section 5.4.3 and the fact that

$$\mathbb{X} = X|T(f) \setminus X|T(\delta(f))$$

$$\forall b \in V_\delta \times \mathbb{X} \bullet P(\mathbb{I}_{\Sigma f}(b)) = P(\mathbb{I}_{\Sigma^{\delta(f)}}(b))$$

which considers only variables whose binding does not exist in δ. Hence, we restrict x to $X|T(\delta(p))$, which allows us to approximate $P(\mathbb{I}'_v(x))$ with 1.

To understand Equation (5.11), consider a patch prioritisation technique that takes a program f, a specification S, and a patch δ as inputs and outputs the probability of the patch being correct with respect to the specification, denoted as $P(\delta(f) \mid S)$. Let Γ denote "leaves bindings probabilistically indistinguishable". We can extend this technique by incorporating the unchanged bindings as follows:

$$P(\delta(f) \mid S) = \mathbb{P}(f,\delta) \cdot P(\delta(f) \mid S, \delta\Gamma) \tag{5.12}$$

We can apply other patch prioritisation techniques to this equation's second term, $P(\delta(f)|S,\delta\Gamma)$, to create a new probability that prioritises patches that probabilistically leave bindings unchanged while eliminating those that do not. Equation (5.12) ensures that we do not prioritise patches that violate variables' properties, such as assigning an expression that could evaluate into an even number to a variable that only takes odd numbers. This equation can be used in combination with existing patch prioritisation to improve their accuracy and eliminate unlikely patches that violate variable properties.

5.5 Implementation

We now detail the interesting design decisions we made in order to realise an analysis framework resting on PSP. First, we discuss the fuzzing tools we used for C and Solidity. We then discuss the exact models and techniques we used to realise Equation (5.3) and estimate unseen program states. We close by describing how we integrated PSP into the Mythril symbolic execution engine and adapted the Trident patch synthesiser to use PSP to prioritise patches.

5.5.1 Logging Fuzzing Information

To realise PSP, we must effectively estimate $P(\mathbb{I}_{\Sigma_f}(\sigma))$. For C programs, we use AFL [92] to generate values bound to program variables, then use these values to define an empirical probability distribution, which we use as our estimate. To log the bindings that the fuzzer observes along with the line numbers, we instrument each variable with a function that records the variable to a global dictionary flushed to stdout at every return/exit point. For smart contracts, we use the proprietary Fuzzer: Harvey [93] to log variable information. We use the solidity events that a program emits. These events trigger the `LOG` opcode and cause the EVM to add entries to the transaction receipt. Although we do not use `LOG`'s transaction receipt, we do use Harvey to capture logs that satisfy the event signature corresponding to capturing the variable's state and write them to a dictionary that we periodically flush to a file. We

can recognise this instrumented event corresponding to capturing the variable's state through its event signature, which is available as input to the `LOG` opcode.

5.5.2 Estimating Unseen States

We use two methods to assign probabilities to unseen program states: a supervised model and an unsupervised statistical method. First, we instantiate the supervised model with Random Forest under the default parameters of scikit-learn. We employ the classic Euclidean distance as the distance function. We instantiate u with 100 and k with 20 for computing Equation (5.3). We use these numbers as they gave good results in some short-scale experiments. For the unsupervised statistical method, we employ two methods: kernel density estimation (with scikit-learn's defaults) and the numerical diffusion method. For numerical heat diffusion, we employ a diffusion factor of 6 and numerically compute it for t=10 seconds. We have observed that it works well on ten assertion samples taken from OpenSSL.

5.5.3 Symbolic Execution

For symbolic execution, we implement PSP's strategy on Mythril [98], a well-maintained symbolic execution engine for smart contracts. We implement this on Mythril's v0.23.22 version, since earlier versions do not strictly adhere to execution timeout due to Z3 not adhering to its prescribed timeout, which will end up tainting the results. We employ a solver-timeout of 25 seconds for all the configurations.

Mythril has two classes of search strategies: ordinary and super strategies. Ordinary strategies can be stacked on top of super strategies, and super strategies are stackable on top of one another. The top strategies in the stack, *i.e.* super strategies, have the highest priority in choosing the next state. In case when the criteria of choice for the super strategy is not satisfied, the choice of the next state goes to the ordinary strategy.

We implemented PSP as an ordinary search strategy. The only super strategy on top of PSP is the `BoundedLoopsStrategy`, which, by default, bounds loops to three iterations. We stack `CoverageStrategy` and `BoundedLoopsStrategy` when running the other baselines on Mythril, such

as the pending constraints search strategy [96] and the default *BFS*. We use the `MythrilCoveragePlugin`, a custom Mythril plugin, to record instruction and branch discovery along with their discovery times. For the threshold θ, we employ $\theta = 0.8$, as that provided the best performance out of all thresholds $\theta \in \{x \mid x \in \mathbb{Z}, 0 \leq x \leq 100, x \equiv 0 \pmod{10}\}$ on a sample of 10 smart contracts taken from SmartBugs-Wild [101] dataset.

5.5.4 Patch Prioritisation

To implement patch prioritisation, we extend Trident's [102] patch synthesiser to use PSP to prioritise patches. Trident enumerates every patch to the depth $d = 4$, checking if each patch matches the specification. We modified Trident to use a priority queue instead, ordered by state probability under PSP's $\mathbb{P}$, defined by Equation (5.11).

5.6 Evaluation

We aim to answer the following questions in our evaluation:

RQ1 *How does* PSP *improve the bug-finding capability of abstract interpretation?*

RQ2 *How does* PSP *improve the bug-finding capability of symbolic execution?*

RQ3 *How does* PSP *improve patch prioritisation?*

We also report on PSP's sensitivity to its two critical hyperparameters, and perform an ablation of its value estimators.

We trained PSP on fuzzing data from coreutils and openssl, split into 90% for training and 10% for testing. We conducted runs on a 16 core, 3.2 GHz machine running Ubuntu 22.10 with 32GB of memory.

To compare PSP with the baselines, we focus on key performance metrics relevant to bug finding tools. First, we present the true positive counts, highlighting the primary objective of bug finding tools — discovering bugs. Second, we report precision, since excessive false positives have a detrimental effect on the usability of bug finding tools [103]. Third, we report the Matthews Correlation Coefficient

Table 5.1: The number of bugs/false positives each tool finds. A_{30} denotes running the AFL fuzzer for 30 minutes. $P(F,\theta)$ denotes running the fuzzer F with threshold θ.

Program		Clam		A_{30}		$P(A_{30},0.5)$		$P(A_{30},0.7)$	
	Bugs	TP	FP	TP	FP	TP	FP	TP	FP
libtiff	14	14	73	1	0	5	8	3	5
libpng	7	7	91	1	0	5	9	5	8
openssl	20	20	173	2	0	8	14	8	11
php	16	16	76	2	0	5	4	5	2
libxml	17	17	83	2	0	7	7	6	4
SQLite	20	20	167	0	0	1	8	0	6
Poppler	22	22	159	1	0	7	16	6	12
Total	116	116	822	9	0	38	67	31	48

Table 5.2: Tool performance using IR measures. A_{30} denotes running the AFL fuzzer for 30 minutes. $P(F,\theta)$ denotes running the fuzzer F with threshold θ.

Tools	Precision	Recall	F_1 Score	MCC
Clam	0.12	1.00	0.22	0.09
A_{30}	1.00	0.08	0.14	0.26
$P(A_{30},0.1)$	0.19	0.42	0.26	0.13
$P(A_{30},0.5)$	0.36	0.33	0.34	0.26
$P(A_{30},0.7)$	0.41	0.28	0.34	0.27

(MCC), which encapsulates our approach's defect classification capacity, considering the full confusion matrix [104]. We also include the F1 Score to facilitate comparison with related work.

5.6.1 RQ1: PSP for Abstract Interpretation

We use the Clam static analyser [94] to overapproximate whether an assertion can be violated and the AFL fuzzer [92] for 30 minutes to under-approximate it. We compare the bugs reported by these tools to the bugs reported by PSP and count the number of false positives or negatives. We sample 7 out of 9 projects from Magma benchmark [95], namely, libtiff, libpng, openssl, php, libxml, Poppler, SQLite. The projects have some explicit assertions that always hold and other assertions that violate to indicate a bug's presence.

Table 5.1 shows the number of false positives and false negatives for various tools; Table 5.2 shows precision, recall, F1 score, and MCC. As an overapproximate analysis, Clam has the highest recall among all tools. Clam also has the lowest precision: it reports 938 bugs, but only 12% of the reported bugs are actually bugs.

In contrast, AFL is under-approximate; it has the highest precision but the lowest recall. After running for 30 minutes, AFL reports only 8% of the 116 bugs.

PSP achieves the highest F1-score, balancing precision and recall. If the threshold value θ is set to 0.5, then PSP finds 38 of the 116 bugs, but it also reports 67 bugs that do not exist. PSP for $\theta = 0.5$ reports substantially more true positives than AFL and substantially fewer false positives than Clam. If we adjust θ to 0.7, the total number of bug reports decreases from 105 to 79, increasing precision from 36 to 41% at the cost of decreasing recall from 33 to 28%.

From Equation (5.2), we observe that PSP only constructs non-zero probabilities for at least one program state *w.r.t.* a statement iff the statement is covered by the fuzzer. For the statements covered by the fuzzer, PSP at $\theta = 0.7$ has correctly identified, by violating an assertion, 77.5% of the buggy statements as bugs. In contrast, the fuzzer has only identified 18.4% of the buggy statements, which it has covered in the span of 30 minutes as bugs.

RQ1: For the threshold $\theta = 0.7$, PSP finds 31 out of 116 bugs and produces 48 false alarms; it finds 22 more bugs than AFL running for 30m (A_{30}) and does so with 774 fewer false positives than Clam.

In summary, PSP outperforms Clam in terms of precision, and outperforms AFL in terms of the number of found bugs, while outperforming the fuzzer on F1 and effectively matching it on MCC. In practice, analyses like PSP are important for software developers who wish to trade finding more bugs against contending with more false positives.

5.6.2 RQ2: PSP for Symbolic Execution

To evaluate PSP's symbolic execution strategy on smart contracts, we compare our strategy (*PSP*) against two baseline strategies: Mythril's default search strategy (*Default*), which is a combination of breadth-first, coverage-directed, and loop bound strategies.[1], and a recent search strategy based on pending constraints [96] (*Pending*), which is implemented on top of Mythril's default search strategy. We opted to

[1]The strategies are called `BFS`, `CoverageStrategy`, and `BoundedLoopsStrategy`.

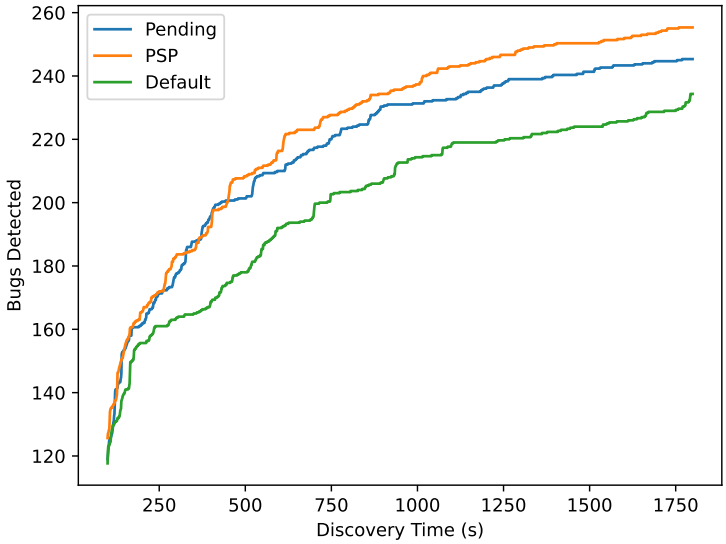


Figure 5.2: Bugs Discovered vs Time.

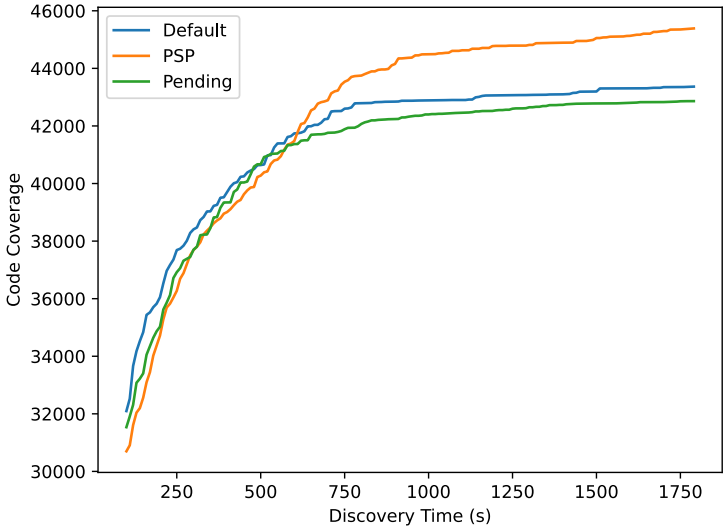


Figure 5.3: Instruction Coverage vs Time.

evaluate PSP using Mythril rather than running KLEE on Magma for homogeneity because KLEE is not robust enough to easily run on Magma and may lack support for certain instructions in this context. In contrast, Mythril is a widely used industry tool which is specifically designed to support symbolic execution on smart contracts, making it both easier to run and more reliable for evaluating PSP's strategy.

We evaluate the symbolic execution strategy of PSP for smart contracts using three sets of 52 random samples extracted from the SmartBugs-Wild [101] dataset, which consists of 47,398 smart contracts from Ethereum mainnet. Smart contracts tend to be smaller than traditional programs due to the limited storage capacity and

high cost of running on the blockchain. The bytecode of a smart contract on EVM has a size limit of 24KB [105]. However, smart contracts operate differently from standard programs because they are state machines, where users can modify the state by triggering one of the smart contract's functions. This potentially leads to infinite execution paths if a user repeatedly triggers a function that modifies the state, which results in complex control flow and an increased number of possible execution paths, even for small programs. The average number of lines of a smart contract in our samples is 379.8.

Figure 5.2 shows the number of bugs found in smart contracts over time by our technique (PSP) and the baseline techniques. Figure 5.3 shows the number of instructions symbolically executed over time. PSP outperforms the baseline (*Default*) and improved strategy (*Pending*) in terms of number of bugs found. In 30 minutes, PSP can find 9% and 4.1% more bugs than *Default* and *Pending*, respectively. In terms of symbolic coverage of program instructions, PSP performs substantially better than *Default* and *Pending* from 10 minutes onwards. For the first 600 seconds, the coverage-guided *Default* and *Pending* strategies perform better. However, the coverage-guided strategies quickly exhaust the easy-to-discover instructions and start underperforming. PSP's search strategy does not explicitly prioritise coverage. Even though it lags initially due to not focusing on coverage, PSP eventually outperforms the other two strategies due to its superior performance once the easy-to-discover instructions are exhausted.

RQ2 (1/2): PSP's search strategy finds 4.1% more bugs in smart contracts than the state-of-the-art pending constrains search strategy.

Abstract symbolic execution (ASE) [97] represents the amenable portion of the symbolic state using value sets and delegates constraint solving partially to cheaper membership tests in these value sets. For an objective comparison with ASE, we implemented the PSP search strategy in the ASE tool for C programs. Since the programs supported by ASE are smaller, the fuzzer thoroughly covers various paths in that programs, enabling a precise estimation for PSP. Thus, our ASE variant does

Table 5.3: The execution time and the number of solver queries for PSP, ASE and default symbolic execution.

Program	PSP		ASE		baseline	
	time(s)	queries	time(s)	queries	time(s)	queries
bubble_sort_1	8.70	0	8.39	0	38.8	67350
bubble_sort_3	21.70	4644	152.80	637821	266.7	1135634
bubble_sort_all	9.30	0	58.75	40320	59.0	234958
dijkstra	17.40	572	156.60	88726	184.7	99564
dirname	0.06	0	0.05	0	74.7	815764
gcd	1.50	0	125.60	3192	123.6	3192
half	0.01	0	0.01	0	16.4	8010
heap_sort_1	0.78	0	0.81	0	1.6	2340
heap_sort_3	0.74	0	22.50	104524	111.3	518822
heap_sort_all	25.30	9674	288.30	960034	111.1	964444
insert_sort_1	7.50	0	7.40	0	38.6	67350
insert_sort_3	24.90	18268	330.00	1420769	650.8	2860000
insert_sort_all	2.30	3244	15.20	80057	18.5	80638
merge_sort_1	0.20	0	0.20	0	0.4	896.0
merge_sort_3	0.30	0	9.40	29460	34.9	147400
merge_sort_all	0.30	0	18.00	78750	18.9	80638
quick_sort_1	4.70	0	4.80	0	4.9	598.0
quick_sort_3	5.90	0	61.40	205666	128.3	446606
quick_sort_all	5.30	0	25.80	78800	26.7	80638
selection_sort_1	4.10	0	4.10	0	559.7	1192300
selection_sort_3	21.50	70831	771.40	2913602	1048.4	3828538
selection_sort_all	2.30	3244	153.20	521344	153.3	526350
kruskal	12.30	1255	179.90	711343	195.4	789714
bellman-ford	18.50	6632	131.70	91752	150.3	102876
binary_search_all	0.03	0	0.03	0	345.0	8000
linear_find_all	0.01	0	0.01	0	32.5	2000
is_permutation	8.20	6055	383.10	1622733	418.0	1716634
loop_invgen	0.10	0	124.80	22859	246.8	34440
min_max_all	0.50	0	30.20	72067	32.8	77706
fibonacci	0.05	0	0.05	0	6.9	206554
outerproduct	0.08	0	0.08	0	45.0	140616

not implement value sets, as they are superfluous on small programs. We employ two thresholds, 0.1 and 0.9, and only solve path constraints if their estimated probability of satisfiability falls into $(0.1, 0.9)$. We simply assume the query is UNSAT for a low threshold (< 0.1) and SAT for a high threshold (> 0.9). We construct data by running fuzzing for 3 seconds. For this result, we use the dataset of small programs employed in the original ASE work [97]. The average program size in this dataset is *ca.* 79 lines of code.

Table 5.3 shows the results for the comparison of the execution time and the number of solver queries against the ASE tool and the default (baseline) symbolic execution on the benchmark set of C programs provided by the ASE authors. Out of 31 samples tested, PSP required solver intervention in only 10 cases, whereas ASE required intervention in 19 cases. In these 10 cases, we observed that the number of queries sent to the solver was at least an order of magnitude lower than the number of queries sent by ASE. Overall, the total number of queries was reduced by 77 fold when using PSP compared to ASE. These results demonstrate that PSP is a promising approach for optimising symbolic execution, reducing the reliance on

solver intervention and improving the efficiency of the technique.

RQ2 (2/2): Compared to ASE, PSP's search strategy reduces the number of solver calls by 77 and the average time of a symbolic execution run by a factor of 15.

5.6.3 RQ3: PSP for Patch Prioritisation

We use the C implementation of Rete Chapter 4 as the baseline for patch prioritisation. We extend Rete with PSP, using the techniques described in Section 5.4.3. We chose the C implementation of Rete because the fuzzing framework for C is more robust when compared to Python. We use MB35 dataset [102], consisting of 35 bugs sampled from ManyBugs [51]. Our evaluation of patch prioritisation involves determining the rank of the correct patch in the ordered sequence of generated candidate patches. We restricted our analysis to 8 bugs, excluding those for which Rete does not generate correct patches within 2 hours, as computing precise ranking for such bugs would be impractical. The average patch ranking for Rete and PSP-Rete (Rete augmented with PSP) are 3999 and 2959, respectively, as shown in Table 5.6.

RQ3: PSP integrated with the state-of-the-art patch prioritisation of Rete helps to decrease the average rank of correct patches by 26% from bugs from ManyBugs.

5.6.4 PSP's Hyperparameters

PSP has two critical parameters: The precision threshold θ and the fuzzer configuration A_{time}. Here, we explore PSP's sensitivity to these two parameters on our subset of the Magma benchmark (Section 5.6.1). The precision threshold of PSP is used for classifying whether a given sample has a bug. Based on how PSP computes probabilities Section 5.3.1, at $\theta = 1$, we have a precision of 1 and at $\theta = 0$, we have a recall of 1. Figure 5.4 presents the F_1 score and MCC as a function of θ. The F_1 score increases rapidly until $\theta = 0.5$, after which it starts to decrease. This drop in F_1 score from 0.9 to 1 is due to the restriction on estimations of probability discussed

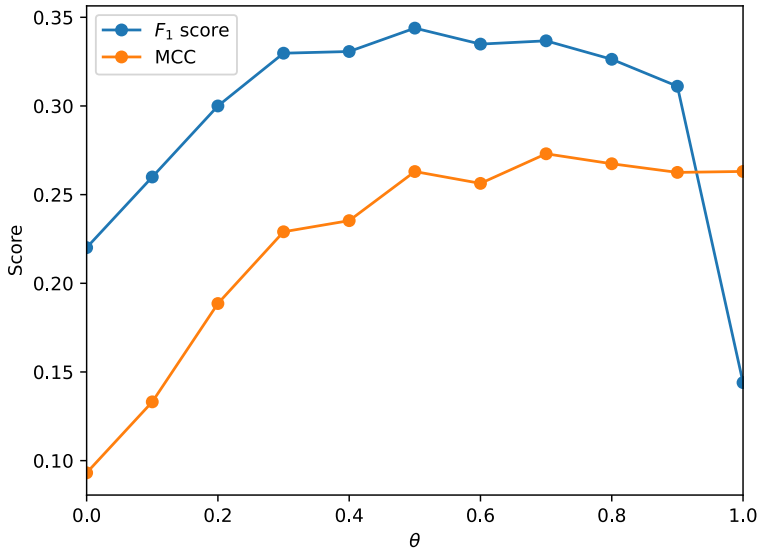


Figure 5.4: This graph shows how PSP's performance varies with θ. Recall that, when $\theta = 0$, PSP conservatively considers all states deemed feasible under an abstract interpretation and, when $\theta = 1$, PSP is equivalent to the fuzzer it is using, and ignores all states the fuzzer did not produce. We see here that performance on F_1 score peaks at just shy of $\theta = 0.5$, whereas MCC peaks at around 0.7.

in Section 5.4.1. Examining Figure 5.4 shows that PSP has the best with MCC at $\theta = 0.7$, and after that, the score decreases.

In Figure 5.5, we plot the Receiver Operating Characteristic (ROC) curve for $PSP(A_{30}, \theta)$ by considering only assertions visited by the fuzzer, as our PSP implementation cannot estimate assertions unvisited during fuzzing. A ROC curve shows the trade-off between the true positive rate and the false positive rate for different classification thresholds. The dotted $x = y$ line represents the performance of a random classifier. The area between a classifier's ROC curve and the dotted line indicates how much the classifier outperforms random guessing. The fact that PSP's ROC curve is well above this line demonstrates that PSP is a decent bug classifier.

The fuzzing configuration A_{time} influences the quality of PSP. A_{time} depends on the fuzzer, which, in our case, is AFL. It also depends on how long the fuzzer runs to generate fuzzing data, which we measure in terms of time and the number of fuzzer iterations. In practice, a user of PSP may wish to identify what fuzzing budget is needed to construct a PSP of sufficient quality. We suggest doing it simply

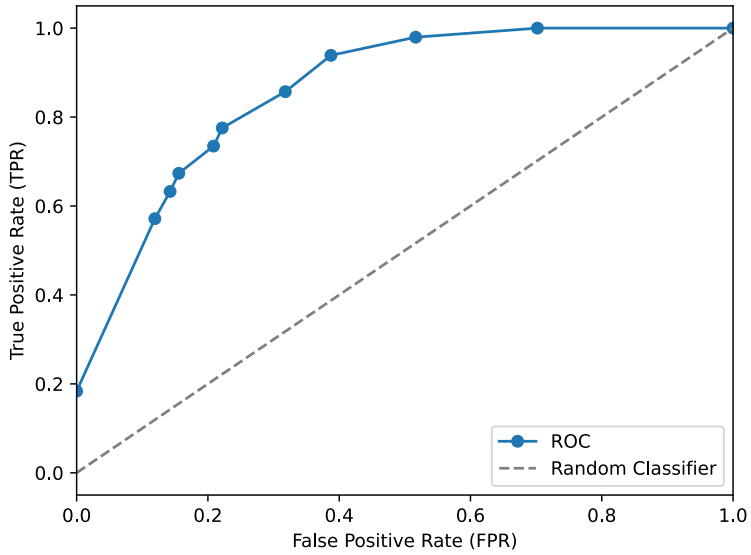


Figure 5.5: This Receiver Operating Characteristic (ROC) curve shows how True Positive Rate and False Positive Rate vary with the threshold. We observe that PSP is strictly above a random classifier in terms of performance.

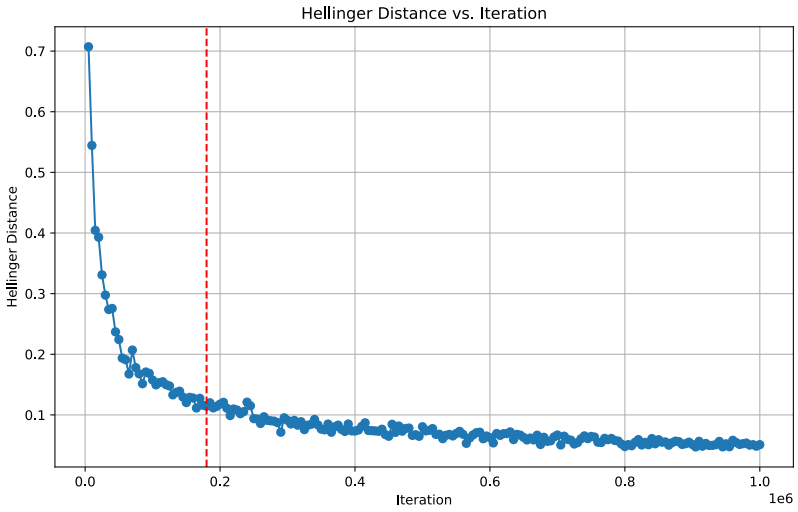


Figure 5.6: This plot illustrates the average Hellinger distance between normalised PSPs across fuzzer iterations with a stride of 5000. The vertical dotted line indicates the number of fuzzer iterations corresponding to a 30 minute fuzzing campaign. We observe that the probabilities indeed converge when using PSP.

by observing the convergence of PSP as the budget increases.

To evaluate PSP's convergence, we computed the difference between PSPs across fuzzer iterations with the stride of 5000. To quantify the distance between the sets of state probabilities of two iterations, we normalise them, and then compute the

Table 5.4: Expanded names of the functions used in Table 5.5

Abbreviation	Expansion
RF	Random Forest
DIF	Diffusing the probability
RFDIF	Random Forest on dense data and DIF on sparse data
RFKS	Random Forest on dense data and KDE on sparse data
DIF	Diffusing the probability using Equation (5.4)
KDE	Kernel Density Estimation

Table 5.5: The performance of estimating techniques from Table 5.4. Coreutils-S denotes the sparse version of Coreutils, and Coreutils-D denotes its dense version.

Dataset	Samples	RFDIF	RFKDE	RF	DIF	KDE
Coreutils-S	100k	99.8%	99.5%	-	99.8%	99.5%
Coreutils-D	100k	99.3%	99.0%	98.4%	62.3%	56.4%

average Hellinger distance [106] between the normalised probabilities as follows:

$$H(p_i, p_{i-5000}) = \frac{1}{\sqrt{2}}\sqrt{\sum_{i=1}^{n}(\sqrt{p_i} - \sqrt{p_{i-5000}})^2}$$

where p_i denotes the probability at the i^{th} iteration.

Figure 5.6 shows how the distance between adjacent PSPs converges with the increase of the number of fuzzer iterations. The vertical dotted line indicates the number corresponding to a 30 minutes fuzzing campaign. The fact that PSP converges implies that we can stop the fuzzer after a certain number of iterations without a significant loss of precision.

5.6.5 Ablation: Estimating Unseen States

Table 5.4 shows various estimating techniques for the unseen program state. We conducted fuzzing on Coreutils by randomly selecting lines for instrumentation, which we divided into two categories: dense and sparse. For the sparse category, we chose samples that did not satisfy the criteria for choosing the supervised model shown in Equation (5.3). We selected variables with a large range of values or samples from states that did not have enough fuzzing samples. For the dense category, we chose variables that satisfied the criteria for running the supervised model, such as error codes, file descriptors, enums, etc. This process resulted in 100k

Table 5.6: Average patch ranking for MB35.

Bug	Rete	PSP-Rete
gmp-a1d3d-f17cb	843	840
libtiff-09e82-f2d98	513	496
libtiff-764db-2e42d	7	6
libtiff-a72cf-0a36d	15842	12039
libtiff-37133-865f7	8566	5711
php-70075-5a8c9	782	725
php-e65d3-1d984	5439	3954
php-63673-2adf5	3	3
Average	3999	2959

samples for sparse and dense data. Additionally, we generated discrete samples (DS) with specific patterns, such as even, odd, modulo, and other randomly generated periodic patterns. For string samples (SS), we extracted them by instrumenting strings from coreutils.

To evaluate the techniques, we considered any probability above 0.5 as indicating the existence of a state and any probability below 0.5 as indicating the non-existence of a state. Table 5.5 reports each technique's performance at estimating unseen states. RF is effective in recognising patterns present when the data is dense. However, for sparse data, both DIF and KDE are effective in estimating the unseen states, although DIF has slightly better results. For techniques such as symbolic execution on smart contracts, RF was rarely triggered (with a trigger rate of $< 1\%$) due to data sparsity. In shot, random forest (RF) outperforms the other methods on dense data; diffusing the probability (DIF) outperforms the other methods on sparse data.

5.6.6 Threats to Validity

We evaluated PSP against Clam using a subset of the Magma benchmarks [95]. Specifically, we uniformly sampled 7 out of the 9 available benchmarks for computational feasibility. The limited size of this sample set may affect the generalisability of our findings to other benchmarks.

Additionally, PSP relies on fuzzing to infer probabilities. Fuzzers like AFL may not cover certain variables or may only observe a biased subset of a variable's possible values, introducing a potential external validity threat.

Our evaluation involved running various tools (Mythril, AFL, Clam, Harvey) with their default configurations. As a result, our findings may not generalize to other configurations or settings beyond those used in this study.

Chapter 6

The Fact Selection Problem in LLM-Based Program Repair

6.1 Introduction

When debugging and fixing software bugs, developers seek bug-related information from a diverse array of sources. Such sources include the buggy code's context, documentation, error messages, outputs from program analysis, etc. Individual pieces of this information, which following recent work we refer to as *facts* [107], have been demonstrated by previous studies to enhance LLMs' bug-fixing efficacy when incorporated into the prompts [108, 109, 110]. Given the ever-increasing context window of cutting-edge LLMs, a critical question emerges: "Which specific facts, and in what quantity, should be integrated into the prompt to optimise the chance of correctly fixing a bug?"

This work is a systematic effort to investigate how to construct effective prompts for LLM-based automated program repair (APR) by composing facts extracted from the buggy program and external sources. We identified 14 atomic facts: those individually studied in the context of APR by previous works, such as the buggy code's context (Prenner *et al.* [111], Xia *et al.* [108], and Chen *et al.* [112]), GitHub issues (Fakhoury *et al.* [109]), and stack traces (Keller *et al.* [113]); angelic values, a semantic fact previously unexplored in the context of LLM-based APR, but that was successfully used for debugging [114] and repair [4]; and those chosen based

on our developer intuition. Since some of these atomic facts are closely related, we combined them into seven compound facts, which from now on we refer to as just facts. Our study was conducted on 314 bugs in open source Python projects from the BugsInPy [64] benchmark.

Our first experiment aims to confirm the utility of the considered facts. Specifically, for each fact, if the potential inclusion of this fact in APR prompts helps to repair some additional bugs, or increases the probability of fixing some bugs. To answer this question, we constructed over 19K prompts tasked to repair the buggy function and containing different subsets of the seven facts for the 314 bugs. Then, we queried an LLM to generate patches, and evaluated the patches using the provided test suites. Finding 1 confirms the utility of each fact, that is each fact helped repair at least one bug that was not repaired by any prompt without this fact. Moreover, all the facts have statistically significant positive impact on the probability of fixing a bug in a single attempt.

Given the utility of each fact, it is tempting to assume that adding more facts always enhances LLM's performance. Contrary to this intuition, Finding 2 reveals that APR prompts are non-monotonic over facts: adding more facts may degrade LLM's performance. An experiment involving 157 bugs showed that prompts incorporating all available facts resulted in 12 fewer bug fixes and exhibited an 8.2% lower probability of repairing a bug within a single attempt compared to the most effective subset. This result aligns with previous research that showed that LLMs do not robustly make use of information in long input contexts [115], and their performance is dramatically decreased when irrelevant information is included into the prompt [116]. Another important consideration is the cost of facts: facts such as the buggy code's context are low-cost to extract, while angelic values or GitHub issues are high-cost, because they require program analysis or human effort.

The non-monotonicity of APR prompts and, simultaneously, the utility of each fact led us to define *the fact selection problem*: determining the optimal set of facts for inclusion in prompts to maximise LLM's performance on given tasks. It can be viewed as a variant of feature selection of classical machine learning for LLM's

in-context learning. We consider two instances of fact selection: *universal fact selection* when the selected facts do not depend on a specific task instance, *i.e.* the bug, and *bug-tailored fact selection* when the fact set is bug-specific.

If there was a universal set of facts that is effective for all bugs, it would significantly simplify the development of LLM-based APR tools. However, our experiments showed that universal fact selection is suboptimal compared to bug-tailored fact selection. Specifically, Finding 3 identified that there is not a universal set of facts that is sufficiently effective, compared to other sets, on all subsets of the 314 bugs. Meanwhile, enumerating sets of facts via *e.g.* greedy strategies while repairing each bug might be impractical, because of the high cost of LLM queries and the necessity to generate multiple responses due to LLMs' nondeterminism. As a practical compromise, we trained a statistical model that we call MANIPLE. It is designed to select facts contingent upon the features of a specific bug. Empirical evidence shows that MANIPLE significantly outperforms a universal fact selection methodology that employs a generic set of facts that exhibited the optimal performance on the training data.

We benchmarked MANIPLE against state-of-the-art zero-shot non-conversational LLM-based APR techniques. On our testing set, MANIPLE repaired 17% more bugs, highlighting the practical impact of the fact selection problem.

The contributions of this work are:

- A systematic study of seven bug-pertinent facts for APR prompts, including angelic values, previously unexplored in the context of LLM.
- An empirical evidence of the non-monotonicity of APR prompts over bug-related facts.
- A motivation and introduction of the fact selection problem for LLM-based APR.
- The first bug-tailored fact selection model, MANIPLE, that significantly outperforms previous related techniques.

```
Please fix the buggy function provided below and output a corrected version.
<.. CoT instructions are omitted ...>

## The source code of the buggy function
```python
# this is the buggy function you need to fix
def read_json(
<... a part of code is omitted ...>
    return result
```
A test function that the buggy function fails
```python
def test_readjson_unicode(monkeypatch):
<... a part of code is omitted ...>
        tm.assert_frame_equal(result, expected)
```
The error message from the failing test
```text
monkeypatch = <_pytest.monkeypatch.MonkeyPatch object at 0x7f567d325d00>
<... a part of message is omitted ...>
pandas/_libs/testing.pyx:174: AssertionError
```
Runtime values and types of variables inside the buggy function
compression, value: 'infer', type: `str`
<... some variables are omitted ...>
lines, value: `False`, type: `bool`

Expected values and types of variables during the failing test execution
path_or_buf, expected value: '/tmp/tmphu0tx4qstest.json', type: `str`
<... some variables are omitted ...>
result, expected value: <...omitted...>, type: `DataFrame`

A GitHub issue for this bug
```text
Code Sample, a copy-pastable example if possible
<... a part of text is omitted ...>
However, when read_json() is called without encoding parameter, it calls built-in
    ↪ open() method to open a file and open() uses return value of locale.
    ↪ getpreferredencoding() to determine the encoding which can be something not
    utf-8
```
```

Figure 6.1: A simplified APR prompt incorporating various facts for fixing pandas:128. ... shows information omitted for brevity. ... shows a part of the GitHub issue that is essential to correctly fix the bug. Too much information in the prompt "distracts" the LLM from the relevant part of the issue description, significantly reducing pass@1 and the correctness rate.

All code, scripts, and data necessary to reproduce this work are available at `https://github.com/PyRepair/maniple`.

6.2 Motivating example

To illustrate the importance of the fact selection problem, we consider the bug pandas:128 [117] in the Pandas data analysis library within the BugsInPy benchmark. This bug arises due to incorrectly handling the default encoding in the

Pandas' `read_json` function. The developer patch for this bug involves setting the encoding to UTF-8 when it is not specified by adding the following lines to the buggy function:

```
+       if encoding is None:
+         encoding = "utf-8"
```

When GPT-3.5-Turbo [118] was prompted to rectify the bug solely based on the source code of the buggy function, it did not successfully address the issue in any of 15 trials. A likely explanation of this failure is that the function itself does not contain any inherently incorrect code, so fixing this bug requires external information.

Drawing upon existing literature on LLM-based APR and relying on our intuition as developers, we assembled a diverse set of bug-related facts to incorporate into the prompt. These include the buggy function's context, the failing test case, the error message, the runtime values of local variables, their angelic values (values that, when taken by program variables during test execution, result in successful passage of the test), and the GitHub issue description. Figure 6.1 gives a simplified representation of the resulting prompt. Adding these facts enabled the LLM to generate a plausible patch, *i.e.* a patch that passes the tests, in four out of 15 trials. However, only two of these patches were correct. The other two hard-coded UTF-8 as the only encoding instead of the default one. The causes of failures to fix the bug included "forgetting" to change the function despite correct chain-of-thought reasoning [119], or hard-coding the encoding inconsistently.

Interestingly, when we removed all facts but the code of the buggy function, the runtime variable values and the GitHub issue, it significantly raised the success rate. Specifically, the LLM generated plausible patches in 12 out of 15 trials, and 11 out of these 12 were correct. A similar high success rate was demonstrated by the prompt with only the buggy function and the GitHub issue. We posit that this is because redundant or irrelevant information in the original prompt "distracted" the LLM from critical details in the GitHub issue (highlighted in Figure 6.1) necessary to repair the bug [116].

We refer to this phenomenon, that adding more facts may degrade LLM's performance, as the *non-monotonicity* of prompts over facts. To provide stronger empirical

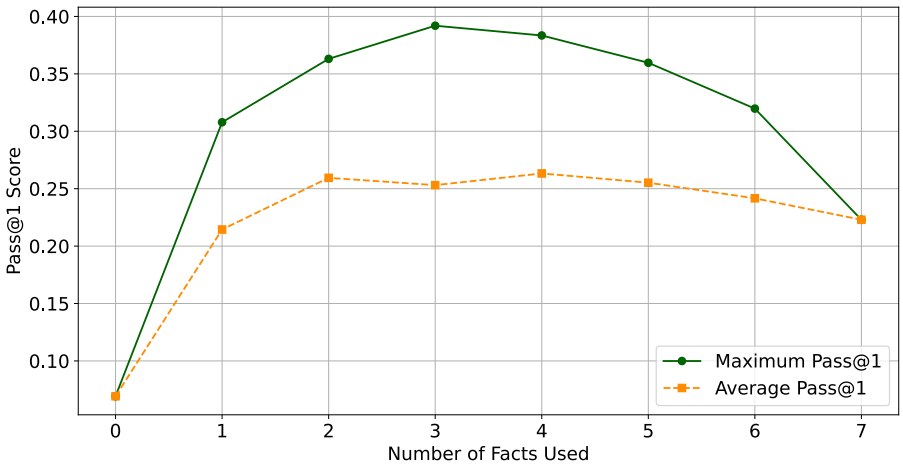


Figure 6.2: Comparison of pass@1 (the vertical axis) for around 10K prompts incorporating subsets of the seven considered facts (the horizontal axis) computed over 15 responses for repairing 157 Python bugs within the BugsInPy benchmark. Zero facts corresponds to the prompt containing only the buggy function without any additional information about the bug. The graph plots the average pass@1 score (dashed orange line) and the maximum pass@1 score (solid green line) across all bugs. This graph clearly shows the non-monotonic nature of APR prompts over facts.

evidence, we conducted a large-scale experiment with 19K prompts containing various subsets of seven facts on 314 bugs in Python projects. Figure 6.2 shows how pass@1, which estimates the probability of generating a plausible patch in a single trial, depends on the number of included facts. The graph shows two lines: the top line corresponds to the scenario when we select the most effective combination of facts for each bug. The bottom line indicates the performance of an average fact set. It is evident that both these functions are non-monotonic.

Apart from the non-monotonicity of prompts, we also discovered that each of the considered facts helps to fix some bugs that cannot be fixed without it. These observations motivated us to formulate the *fact selection problem*, the problem of selecting facts for a given bug to maximise the chance of repairing it and propose a model MANIPLE that selects effective facts based on features of a given bug.

6.3 Study Design

This section discusses the experimental setup, the facts chosen for our study, and how they are represented in prompts.

6.3.1 Experimental Setup

We denote the set of bugs as B and the set of all bug-relevant facts as $\mathbb{F}$. Since we aim to investigate how using various facts impacts the success of APR, we define the set of all jobs as $\mathbb{J} = B \times 2^{\mathbb{F}}$, which pairs bugs with sets of facts.

Zero-shot Prompting for APR An advantage of large language models, such as ChatGPT [118], is they can be adapted to a downstream task without retraining via *prompt engineering* [120]. A prompt refers to the input or instruction given to the model to elicit a response. Prompting typically takes either the form of *zero-shot*, *i.e.* directly providing the model a task's input, or *few-shot* [121], where the model is provided with a few examples. In this work, we investigate a zero-shot APR approach. Although the few-shot approach is promising, it requires finding high-quality examples [122], which we leave for future work.

Function-granular Perfect Fault Localisation APR tools repair bugs by first localising suspicious locations. For an objective evaluation of APR tools, Liu *et al.* [74] argues for the use of *perfect fault localisation (PFL)*, that is, when the buggy locations are known to the tool. PFL can resolve to difference granularity levels: notably, the line or the function. Liu *et al.* argues that fault localisation tools do not offer the same accuracy in identifying faulty locations at different granularities, "making function-level granularity appealing for limiting unnecessary trials on fault positive locations". We found that, in BugsInPy [64], the average Ochiai [18] rank of the buggy line is 2502 and the buggy function is 22 across the entire codebase, making it much more likely to localise the buggy function than the buggy line. Apart from that, although 70% of the bugs in BugsInPy require modifying only a single function, 65% of them modify multiple lines within this function. Localising a bug to multiple lines is harder than targeting individual lines. Meanwhile, cutting-edge models, like GPT-3.5-Turbo, effectively fix bugs even without specifying the exact lines, *i.e.* when only the buggy function is provided. Thus, in contrast to some previous

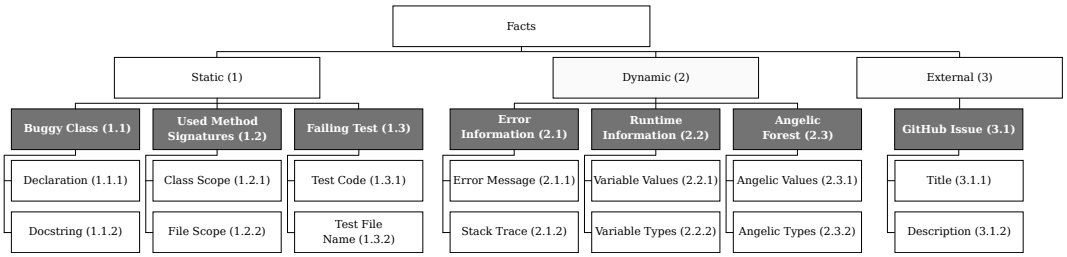


Figure 6.3: This work uses seven compound facts (dark rectangles) across three categories for constructing program repair prompts. Each fact is composed of two atomic facts. Each prompt contains the buggy function to be repaired; the facts can be included based on the employed fact selection strategy.

studies [12, 46, 108], we use function-granular PFL.

Python Bug Benchmark APR tools are typically compared on datasets of bugs extracted from real world projects, such as Defects4J [123] for Java and BugsInPy [64] for Python. In this work, we use BugsInPy because of Python's ever-increasing importance and popularity. BugsInPy contains 501 bugs from 17 popular Python projects such as Pandas [124] and Matplotlib [125]. Among them, we selected a subset of 314 bugs, which we refer to as **BGP314**, that require modifications within a single function, due to our PFL approach, and that we were able to reproduce. To investigate APR performance on various classes of defects, we consider three parts of BGP314:

- **BGP157PLY1**: This dataset comprises 157 bugs, which have been uniformly selected from BGP314 for the purpose of training and analysis.
- **BGP157PLY2**: Consisting of 157 bugs, this dataset is the complement of BGP157PLY1 in BGP314. Used for evaluating fact selection strategies.
- **BGP32**: A subset of 32 bugs uniformly sampled from BGP157PLY1, intended for preliminary studies on finding parameters, such as determining the right response count to reduce the variance.

LLM Nondeterminism Nondeterminism in LLMs leads to varying outcomes between trials, which poses a challenge for analysing results [126]. To alleviate it, we use the pass@k measure, which represents the probability that at least one query

out of k will be successful in solving a problem. Previous work [39] recommends estimating pass@k as

$$\text{pass@k}(J) \triangleq \mathbb{E}_J \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{6.1}$$

where $\mathbb{E}_J$ denotes expectation over the set of jobs J, n is the number of responses obtained from the LLM, where $n > k$ and c is the number of successes found in the n responses. Larger n helps reduce variance. A pilot study using BGP32 reveals that when $n = 15$ and $k = 1$, pass@k exhibits the average standard deviation of *ca.* 0.04, and that further increasing n only marginally decreases standard deviation. This is discussed in detail in supplementary materials (Section 6.8).

For generate-and-validate APR [127] that operates by iteratively generating and testing patches until it finds one that passes the tests, a commonly used measure is the number of bugs for which at least one patch passes the tests among LLM's responses:

$$\text{\#fixed}(J) \triangleq |\{b \mid j \in J, c_j > 0\}| \tag{6.2}$$

where $j = (b, F)$ is a job, and c_j is the number of responses that pass the tests for the job j.

Test-overfitting in Program Repair APR techniques repair bugs *w.r.t.* correctness criteria, such as tests or formal specification. Since tests do not fully capture the intended behaviour, automatically generated patches based on tests may be incorrect [128]. Thus, the APR literature distinguishes between *plausible* patches, patches that pass the tests, and *correct* patches, patches that satisfy the intended requirements. Since manually labelling a large number of patches is resource-intensive and error-prone, most analyses of the fact selection problem with pass@k and #fixed in this chapter count plausible patches as successes. We only label correct patches when comparing our tool with other APR techniques in Section 6.6.

In our experiments, we used the latest version of GPT-3.5-Turbo, gpt-3.5-turbo-0125, which has a 16K context window. As of March 2024, the cost of reproducing the experiments in this chapter using OpenAI API [129] is $479.

6.3.2 Bug-Related Facts

The considered facts were collected from previous LLM-based APR research, previous non-LLM-based APR literature, and our intuition as developers. In total, we collected 14 atomic facts, but since many of them are related, we grouped them into seven compound facts, which we refer to simply as facts or $\mathbb{F}$. These seven facts are divided into three categories: static, dynamic, and external, as shown in Figure 6.3.

Buggy Class (1.1) The declaration of a class containing the buggy function provides a broader context and dependencies. A class docstring offers insight into the overall purpose and functionality of the class.

Used Method Signatures (1.2) Considering methods used within the buggy function, as shown by Chen *et al.* [112], allows for the analysis of dependencies and potential side effects that might contribute to the incorrect behaviour.

Failing Test (1.3) The code of a failing test, as shown by Xia *et al.* [110] provides useful context for repairing a buggy function as it specifically highlights the conditions under which the program fails.

Error Information (2.1) Previous approaches showed that using error messages [110] and stack traces [113] improves LLM's bug-fixing performance.

Runtime Information (2.2) Runtime values and types of the function's parameters and local variables during the failing test execution provide an LLM with concrete data about the program's behaviour.

Angelic Forest (2.3) For a given program location, a variable's *angelic value* [114] is a value that, if bound to the variable during the execution of a failing test, would enable the program to pass the test. Angelic forest [4], previously applied for synthesis-based repair, is a specification for a program fragment in the form of pairs of initial states and output angelic values, such that if the fragment satisfies these pairs, then the program passes the test.

Inspired by this approach, we added a variant of angelic forest to a prompt; this variant combines variable values at the beginning of a function's execution coupled with the angelic values at the end of a function's execution, *i.e.* the input/output requirements of the function. Since Python is dynamically typed, we specify both

the values and types of variables. Angelic values can be computed using symbolic execution [4, 114]; however, due to the immaturity of Python symbolic execution engines, we were unable to execute them on bugs in BugsInPy. Thus, we extracted angelic values from the correct versions of the programs via instrumentation.

GitHub Issue (3.1) A GitHub issue, when available, provides important contextual information for fixing the bug, as shown by Fakhoury *et al.* [109].

To denote subsets of $\mathbb{F}$, we utilise seven-width *bitvectors*, where the i-th bit indicates whether the i-th compound fact in our taxonomy (Figure 6.3) is included in the set. For example, 0000100 corresponds to the set containing only the runtime information (2.2).

6.3.3 Prompt Design

We construct prompts via the prompt engineer $E\colon B \times 2^{\mathbb{F}} \to \Sigma^*$, which builds a prompt over the alphabet Σ to repair an input bug using a subset of facts from $\mathbb{F}$. The prompt is constructed with the directive “Please fix the buggy function provided below and output a corrected version" along with the included subset of facts. The buggy function’s code, together with its docstring, is provided as part of the prompt for the LLM to effectively fulfill this directive. Each fact is incorporated via a specialised prompt template. Figure 6.1 shows an example prompt with incorporated facts, and the fact templates are detailed in supplementary materials (Section 6.9).

We employed the standard chain-of-thought [119] approach by instructing the LLM to reason about the provided facts, as detailed in supplementary materials (Section 6.10).

In our preliminary experiments, we discovered that LLM often generates incorrect import statements, which makes it hard to automatically extract patches from the responses and apply them to the code. To address it, we explicitly added the import statements in the current file to the prompts. A small study showed that this consistently improves the success rate, as detailed in supplementary materials (Section 6.11).

6.4 The Fact Selection Problem

Let an LLM be a function from a string, *i.e.* a prompt, to a set of strings, the responses R. We consider an arbitrary measure $m : 2^R \times 2^C \to \mathbb{R}$ that scores a set of LLM responses *w.r.t.* some correctness criteria C that maps correct patches to a high score and incorrect patches to a low score, and a prompt engineer $E : B \times 2^{\mathbb{F}} \to \Sigma^*$.

Definition 18 (Fact Selection Problem)**.** *Given a set of bug-relevant facts $\mathbb{F}$, a prompt engineer E, a buggy program b, and correctness criteria for that buggy program C_b, the* fact selection problem *is to find $F \subseteq \mathbb{F}$ that maximises*

$$\arg\max_{F \in 2^{\mathbb{F}}} m(LLM(E(b,F)), C_b) \tag{6.3}$$

C_b encompasses any of the standard correctness criteria such as a test suite or a specification, *etc.* $\overset{*}{F}_b$ denotes an optimal solution of Equation (6.3) for b; We use $\overset{*}{F}_b(\mathbb{F})$ to denote the optimal F for the buggy program b over the fact set $\mathbb{F}$.

Similarly, we use $\overset{*}{F}_B(\mathbb{F})$ to denote the optimal fact set F over all the buggy programs $b \in B$ and the fact set $\mathbb{F}$. This can be defined as the solution to the equation below.

$$\arg\max_{F \in 2^{\mathbb{F}}} \sum_{b \in B} m(\text{LLM}(E(b,F)), C_b)$$

We aim to answer the following questions in this section:

- *How does the inclusion of each fact affect the overall effectiveness of a program repair prompt?*
- *Is there point beyond which adding facts to a program repair prompt degrades its performance?*
- *Can a fixed subset of facts be universally optimal up to a tolerance of ε for bug resolution across various bug sets?*

The first question examines the impact of each fact on the repair effectiveness, questioning whether every fact contributes positively to the resolution process. Section 6.4.1 answers this question affirmatively, showing the inherent value of each

fact. The second question delves into the potential for diminishing returns or even detrimental effects from overloading a prompt with too many facts, suggesting an optimal threshold for fact inclusion that maximises prompt efficacy. If there is no such point, then the optimal strategy will be to include all the facts. Section 6.4.2 shows that, on our dataset, adding facts to a prompt is non-monotonic. Formally, the final question asks whether, $\forall b \in B$, the following equation holds:

$$m\big(\text{LLM}(E(b, \overset{*}{F}_B)), C_b\big) = m\big(\text{LLM}(E(b, \overset{*}{F}_b)), C_b\big) + \varepsilon \tag{6.4}$$

This equation asks whether one can select a fact set for a set of bugs that is as effective as a fact set tailored to each bug. Section 6.4.3 answers this question by showing that this statement does not hold. The above answers, combined, establish the importance of the fact selection problem.

6.4.1 Fact Utility

A fact should only be considered for inclusion in a prompt if it has a potential to improve the outcome. To confirm the utility of the considered facts $\mathbb{F}$, we simplify the premise by assuming that facts are independent and pose two questions: "What is the utility of each individual fact in improving repair performance on our dataset if we select the most effective fact set for each bug?" and "What is the utility of each individual fact in improving repair performance on our dataset if we select a random fact set for each bug?" The first question addresses the potential effectiveness when we precisely know which facts to choose for a specific bug. This notion of utility, which we refer to as *utility under optimal fact selection*, is relevant when we have a method to closely approximate an optimal solution $\overset{*}{F}_b(\mathbb{F})$ to Equation (6.3), *i.e.* choose the most effective facts for each bug $b \in B$. The second question explores the expected outcomes when we lack specific knowledge about which facts to select, and hence make a random choice. This notion of utility, which we call *utility under uniform fact selection*, is relevant when solving Equation (6.3) is either difficult or infeasible.

To estimate the utility of each fact, we generated prompts containing all subsets

Fact	# Excl. Fixed	Gain	Shapley
Error Info.	9	0.48	0.54
Angelic Forest	7	0.08	0.11
Failing Test	7	0.06	0.08
Used Method S.	4	-0.12	-0.18
Buggy Class	4	-0.03	-0.05
GitHub Issue	3	0.44	0.51
Runtime Info.	2	0.03	0.05

Table 6.1: On the left side, this table reflects the number of bugs ("# Excl. Fixed") that could only be resolved by incorporating the specific fact into the repair process under an optimal fact selection for BGP157PLY1. On the right side, we report both "Gain" as defined by $A(f)$ in Equation (6.5) and Shapley values under a uniform fact selection. The "Gain" cells quantify the average percentage increase in prompt repair performance from adding the fact. "Error Information" is the most effective and "Used Method Signatures" the least. All the Shapley values are scaled by a factor of 16.

of the considered 7 facts (the remaining one, the buggy function, is always present in the prompt), which resulted in 19228 prompts in total for BGP314, which is below 314×2^7 since some facts are not available for some bugs. For each prompt, we computed 15 responses to estimate the measures pass@1 and #fixed. This enabled us to both evaluate an optimal selection strategy by explicitly considering $\overset{*}{F}_b$ for each bug, and a uniform selection strategy.

To show the utility of facts $\mathbb{F}$ under optimal fact selection, we compute two measures. First, we compute the number of bugs that were fixed only when the fact is available, presented in the column "# Excl. Fixed" of Table 6.1. Notably, the inclusion of "Error Information", "Angelic Forest" and "Failing Test" each leads to fixing the most additional bugs that cannot be fixed without them. Importantly, the table underscores the bug-fixing utility of every fact that we consider by showing that it exclusively resolves a positive number of bugs.

Second, we analysed each fact's utility under optimal selection by how its inclusion in or its exclusion affects pass@k. We do so by simulating the scenario where specific facts are missing: If a fact f were missing, we would be forced to compute $\overset{*}{F}_b$ over $\mathbb{F} - \{f\}$ for each bug $b \in B$. The baseline for this scenario is when all the facts are available.

Each row in the table in Table 6.2 details the pass@1 attainable by the best

Fact f	With f	Without f	Δ	p-value
Error Info.	0.403	0.331	0.071	0.00000
GitHub Issue	0.403	0.341	0.062	0.00000
Angelic Forest	0.403	0.371	0.032	0.00000
Failing Test	0.403	0.375	0.027	0.00007
Runtime Info.	0.403	0.384	0.019	0.00001
Buggy Class	0.403	0.392	0.010	0.00028
Used Method S.	0.403	0.393	0.009	0.00327

Table 6.2: With f reports the pass@k over BGP157PLY1 of the fact **f** labelling the row; **Without f** is pass@k without using **f**; and Δ shows their difference. Removing facts does indeed decrease the performance of the best performing prompt across all the facts.

prompts, with and without a specific fact (denoted by f). For each bug $b \in B$, prompts are constructed using optimal fact sets $\overset{*}{F}_b(\mathbb{F})$ over all facts and $\overset{*}{F}_b(\mathbb{F} - \{f\})$, excluding the fact f. The consistent reduction in pass@1 (denoted as Δ) emphasises the value of each fact in bug fixing.

To determine the statistical significance of the impact of each individual fact under optimal selection, we calculated pass@1 scores for all bugs, both with and without a particular fact. These scores were then compared. The Wilcoxon signed-rank test shows that the inclusion of each fact has a statistically significant effect on each bug's optimal fact set.

We evaluate the utility of individual facts under uniform selection using two complementary measures: Shapley [130] and a new measure we introduce and call *fact gain*, defined in Equation (6.5). We report fact gain along with Shapley for two reasons: (1) Shapley's results are hard to interpret and (2) we have the luxury of exhaustive enumeration, since our fact set is small. Fact gain computes the net increase in pass@1 scores due to the addition of a specific fact f; we defined it by adapting relative change [131] to our problem domain by setting the reference value to fact subsets that do not contain the measured fact f. We define the gain of each fact as follows:

$$A(f) = \frac{\text{pass@k}(J_f) - \text{pass@k}(J_{\overline{f}})}{\text{pass@k}(J_{\overline{f}})} \tag{6.5}$$

Here, $J_f = \{(b,F) \in J \mid f \in F\}$ is the subset of problems where the fact set F includes the fact f, whereas $J_{\overline{f}} = \{(b,F) \in J \mid f \notin F\}$ encompasses those that exclude the fact f. $A(f)$ computes the aggregate gain from incorporating the fact f into the repair process, effectively measuring the fractional increase in the aggregate likelihood of achieving a successful repair when the fact is added.

Table 6.1 showcases the significance of each fact in APR prompts, leveraging both aggregate gain ($A(f)$) and Shapley values, as defined in Equation (6.5). An interesting observation from the experimental results in Table 6.1 is that the facts "Buggy Class" and "Used Method Signatures" exhibits a negative aggregate gain, indicating its inclusion might adversely affect the repair outcome on average. Meanwhile, if we select the most effective facts set for each bug, each of these facts enables LLM to fix 4 additional bugs that were not fixed without it. This shows the importance of fact selection.

Finding 1. *Under the assumption that we select the most effective fact set for repairing each bug, each of the considered facts demonstrates its utility on our dataset.*

If we select an optimal fact set for repairing each bug, then each of the considered seven facts demonstrates its utility on our dataset: having it as an option for inclusion into the prompt helps to repair at least one bug exclusively, and has a statistically significant positive impact on pass@1. When we select facts uniformly for each bug, five of the seven facts show utility, as having them as options increases the probability of fixing a bug.

6.4.2 Impact of Fact Set Size on Prompt Performance

In this section, we investigate the concept of prompt monotonicity by examining how the incremental addition of facts affects prompt performance. Monotonicity, in this context, refers to a consistent improvement in performance with each additional fact. This implies that more information invariably leads to better outcomes. Conversely,

non-monotonicity indicates that there exists a threshold beyond which adding more facts does not enhance, and may even degrade, performance.

Similarly to Section 6.4.1, we evaluate the non-monotonicity of prompts in two settings: under an optimal fact selection and under a random selection. For each of them, we aggregate pass@1 scores for all bugs over sets containing a varying number of facts. Figure 6.2 presents the performance of pass@1 for the prompts across different fact set cardinalities, ranging from 0 to 7. The maximum pass@1 corresponds to an optimal fact selection for each bug, and the average pass@1 corresponds to a random fact selection.

Observing the trend lines, it is clear that the maximum pass@1 score generally rises with an increasing number of facts, peaking at fact sets containing 3 elements before a sharp decline, and the average pass@1 score exhibits a plateau from 2 to 5 facts and subsequent drop-off beyond this point. This confirms the non-monotonicity of prompts.

Finding 2. *Prompt performance is non-monotonic with respect to the number of included facts. While adding facts generally improves performance, there exists a threshold beyond which additional facts hinder performance.*

6.4.3 Non-Existence of a Universally Optimal Fact Set

A "universally optimal fact set" in the context of automated program repair is a collection of facts that, when applied, yields the highest effectiveness in terms of bug fixes and pass@1 scores across a wide range of projects. For defining a universally optimal fact set, we first define the quality of a fact set in terms of the following properties:

- **Efficiency:** A fact set is efficient when it outperforms alternative fact sets.
- **Universality:** A fact set is universal when it is efficient up to ε-tolerance, as defined in Equation (6.4).

Project	Best Fact Set	Pass@1	
		Project	Total
luigi	0011111	0.53	0.26
black	0010111	0.38	0.22
fastapi	0000101	0.19	0.18
httpie	0001111	0.50	0.25
pandas	0011001	0.25	0.25
tornado	0011000	0.37	0.20
ansible	0010101	0.10	0.12
matplotlib	0001001	0.36	0.25
cookiecutter	0001101	0.93	0.20
tqdm	0001111	0.00	0.24
youtube-dl	0011001	0.08	0.25
keras	0101111	0.32	0.23
scrapy	0011111	0.45	0.26
sanic	0010101	0.64	0.21
thefuck	0001001	0.16	0.21

Table 6.3: Comparison of project-specific best fact sets and their project's average Pass@1 scores and the total average Pass@1 scores across all projects in BGP157PLY1. The fact sets are represented using their bitvector encodings.

- **Coverage:** The set of bugs a fact set can resolve.

We define the function $Coverage : 2^{\mathbb{F}} \rightarrow 2^{B}$ where $F \subseteq \mathbb{F}$ is a fact set and B is the set of all bugs in the dataset being considered, such that $Coverage(F)$ returns the set of bugs fixed by the fact set F.

The Coverage Ratio for a given fact set F is defined as:

$$CR(F) = \frac{|Coverage(F)|}{|\bigcup_{\forall F_i \subseteq \mathbb{F}} Coverage(F_i)|} \tag{6.6}$$

We prefer sets that maximise universality and coverage ratio.

Table 6.3 showcases the best performing fact sets for each project alongside their project-specific and overall training dataset Pass@1 scores. Notably, no single fact set dominates across all projects; for example, the fact set 0010101 leads in the Sanic project with a Pass@1 of 0.21, falling short of the maximum Pass@1 score of 0.26. Similarly, for FastAPI, the best fact set, 0000101, secures a project-specific Pass@1 of 0.19, which is more than its overall score of 0.18 but significantly below the highest Pass@1 of 0.26. Moreover, the table reveals that the highest occurrence

Approach	#fixed
Best Fact Set in BGP157PLY1	77
Best Fact Set in BGP157PLY2	84
All Facts	72
Top 5 Union	99
Total Union	119

Table 6.4: Comparison of various approaches based on the number of bugs plausibly fixed in BGP157PLY2. Note that the "Best Fact Set in BGP157PLY2" puts an upper bound on universal fact selection for BGP157PLY2.

count for any fact set is merely 2, with five fact sets appearing twice. This distribution underscores the variability in fact set effectiveness across projects, casting doubt on the possibility of identifying a universally optimal fact set that maintains superior performance across the broad spectrum of repositories and bugs.

Table 6.4 assesses the effectiveness of various approaches of fact selection in producing plausible fixes. The analysis underscores the Best Fact Set, identified as 1111001, which was selected for its highest coverage in terms of the number of bugs it could fix according to the training data, compared against broader approaches such as the Top 5 Union and Total Union. The Top 5 Union, which aggregates the bugs fixed by the top five best performing fact sets, generates fixes that pass tests for 99 bugs, while the Total Union, encompassing bugs fixed by all fact subsets, resolves 119 bugs. These unions significantly surpass the Best Fact Set in bug resolution capability, fixing many additional bugs (22 and 42 respectively). This shows that the highest coverage ratio of the fact set is $CR(1111001) = 0.65$ that it fixes 65% of the bugs while missing 35% of the bugs fixable by other sets.

These results highlight that the Best Fact Set does not have a high coverage ratio, especially compared to the theoretical maximum of an optimal fact selection which has a coverage ratio of 1. The coverage ratio's delineation as monotonic — in that the fact set with the highest number of bugs fixed is deemed the best — indicates that within this dataset, no fact set achieves a high coverage ratio. Table 6.4 further shows the limitations inherent in static fact selection strategies, as demonstrated by the performance of the Best Fact Sets *w.r.t.* BGP157PLY2. These sets fix 84 bugs, and set the upper limit for universal fact selection in BGP157PLY2. These sets

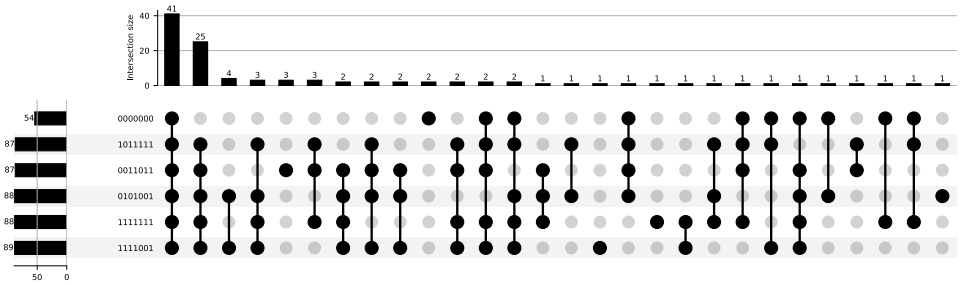


Figure 6.4: Presented above is an upset diagram contrasting the top 5 fact sets from BGP157PLY1 along with the standard fact set which includes the buggy function and chain-of-thought instructions, encoded as the bitvector 0000000. The diagram highlights the intersection sizes at the top, with the most substantial intersection being represented by 41. We can observe that these fact sets individually fix up to a total of 7 bugs. The total number of bugs fixed by these 6 fact sets is 107, which is 18 more than the best fact set on this dataset which fixes 89 bugs.

were not ranked high in the training data, as they were positioned at ranks 16 and 34, respectively out of 128 candidates.

The UpSet diagram [132] in Figure 6.4 shows the combinatorial overlap among the top 5 fact sets from BGP157PLY1, as well as a baseline which does not contain any facts, encoded with the bitvector 0000000. The diagram is particularly instructive in revealing the number of bugs addressed by various intersections of these fact sets, with the largest subset intersection resolving 41 bugs. Each of the fact sets is shown to individually contribute to the resolution of up to 7 bugs. Cumulatively, these 6 fact sets fix a total of 107 bugs, surpassing the efficacy of the single best fact set, which fixes 89 bugs. This UpSet plot helps us in answering the question of the existence of a universal fact set *w.r.t.* the number of bugs fixed, we would expect to see a row with dots in most, if not all, columns, signifying its presence in the majority of intersections. However, the absence of such a pattern in this upset plot indicates there is no single fact set that fixes all bugs. Instead, different fact sets are effective for different bugs, and no single set appears as a common element across all or most bug fixes.

Finding 3. *The diversity in the best fact sets across different repositories (Table 6.3), and the significant difference in the bugs fixed by the top five fact sets*

(Figure 6.4), both point to the absence of a universal fact set in our dataset.

6.5 Selecting Facts with MANIPLE

Universal fact selection, as demonstrated in Section 6.4.3, does not achieve consistent performance across all the bugs in our dataset. Thus, to automate creating bug-tailored prompts, we introduce MANIPLE, a random forest trained to select relevant facts for inclusion in the prompts.

We focus on the task of predicting the success or failure of test executions based on vectors representing features extracted from both the prompt and the code.

Our training dataset $\mathscr{D}$ is constructed from the BGP157PLY1. It consists of pairs (j, y), where:

- $j = (b, F) \in \mathbb{J}$ represents a job, which is a tuple consisting of a bug b along with and a fact set $F \subseteq \mathbb{F}$
- $y \in [0, 1]$ represents the probability of successfully fixing the bug b in a single trial, given the fact set $F \subseteq \mathbb{F}$. This probability is computed using pass@1.

We manually craft a set of features $\mathbf{f}(j)$ based on domain knowledge and the characteristics of the facts. The feature function $\mathbf{f} : \mathbb{J} \to \mathbb{R}^m$ maps the input job $j \in \mathbb{J}$ to an m-dimensional feature space.

The goal is to train a machine learning model $\mathscr{M}$ capable of using the feature vector $\mathbf{f}(j)$ to accurately predict the probability of success. The model is designed to minimize the prediction error over the dataset using a loss function L, as follows:

$$\arg\min_{\mathscr{M}} \sum_{(J,y)\in\mathscr{D}} L\left(\mathscr{M}(\mathbf{f}(J)), y\right),$$

where $L\left(\mathscr{M}(\mathbf{f}(J)), y\right)$ measures the loss between the predicted probability $\mathscr{M}(\mathbf{f}(J))$ and the true probability y. Here, J represents the set of facts, y is the actual probability of success, and $\mathbf{f}(J)$ is the feature vector derived from J.

The choice of loss function L depends on the nature of the task. For example, in regression tasks, which use distance functions, L may be the Mean Squared Error, which measures the squared differences between predicted and actual values. In classification tasks, Cross-Entropy Loss is commonly used, as it quantifies the difference between the predicted probability distribution and the true distribution.

This is achieved through an appropriate training process that adjusts the parameters of $\mathscr{M}$ based on the training data $\mathscr{D}$.

Using this model $\mathscr{M}$, we can select the optimal fact set by choosing the one that yields the highest predicted probability of success according to the model's output.

6.5.1 Feature Selection

To train a machine learning model to meet these objectives, we define the feature vector, $\mathbf{f}(\mathbf{x}) = [\mathbf{b}, rep\_id, \ell, c]^T$, where

- Bitvector (**b**): encodes the fact set F, where $\mathbf{b} \in \{0,1\}^n$.
- Repository ID (rep_id): uniquely identifies the source repository of the bug b.
- Prompt Length (ℓ): the length of the prompt in either characters or tokens, where $\ell \in \mathbb{N}^0$. Cross-validation determines whether characters or tokens are chosen.
- Cyclomatic Complexity (c): a measure of code complexity that quantifies the number of linearly independent paths through a program's source code.

This choice of features was guided by the investigation conducted on BGP157PLY1. Specifically, the inclusion of the Repository ID (*rep_id*) is supported by the findings in Table 6.3, which show significant variability in the optimal bitvectors for fact selection across different projects. This variability underscores the influence of the repository context on successful repair. However, this may limit the classifier's performance with respect to *rep_id*'s it has previously encountered, as this feature is not generalisable across unseen repositories, unlike the rest of the features. Additionally, Prompt Length (ℓ) was identified as a crucial factor, displaying a Spearman correlation of -0.18 with the pass@1 for repair success,

accompanied by a highly significant p-value of $p < 10^{-129}$. This correlation holds for both token and character lengths of prompts, indicating that an increase in prompt length is associated with a decrease in LLM performance — a conclusion that is further supported by the non-monotonicity of adding facts to prompts, as noted in Figure 6.2, as more facts increase prompt length.

Cyclomatic Complexity emerged as another pivotal feature, showing a negative Spearman correlation of -0.1 with the pass@1, p-value of 10^{-42}. Unlike prompt length, Cyclomatic complexity does not depend on the fact set chosen (recall that the buggy code itself is necessarily always included) but remains instrumental in predicting the pass@1 for a bug, mainly for scenarios where no prompt generates a successful fix.

6.5.2 MANIPLE: A Random Forest for Fact Set Selection

Leveraging these observations, we present MANIPLE, a random forest model (Section 2.3.1) for the fact selection task and evaluate it in both regression and classification settings. As a regressor, the model directly predicts the pass@1 for each job. In the classification task, we categorise the scale (i.e., [0, 1]) based on the number of fact sets considered by the model. Our analysis reveals that classification performs better. This finding can be attributed to the noise and significant variance present in our data, as detailed in supplementary materials (Section 6.8). The classification method proved more robust to variance in pass@1 compared to regression. Additionally, ordering and ranking the fact sets did not yield comparable performance. This is likely due to the variance in ranks, which directly stems from the variance in pass@1.

Additionally, we trained MANIPLE only on the top five highest-performing fact sets. These sets were identified using bootstrap aggregation, where we evaluated the performance of each fact set across multiple bootstrap samples drawn from our dataset. By aggregating the outcomes, we identified the best performing fact sets. We optimised the model's hyperparameters through a comprehensive grid search, evaluating the performance of each combination of parameters. To ensure the model's generalisability and to prevent overfitting, we employed k-fold cross-validation with $k = 5$.

6.6 Comparing MANIPLE with SOTA LLM-Based APR

We compared MANIPLE with existing zero-shot non-conversational LLM-based APR methods that incorporate various types of information into prompts. Our baselines include approaches whose prompts include the following facts:

- Buggy Function only, denoted as T_0
- Buggy Function, Buggy Class and Used Method Signatures, denoted as T_1, an approach similar to the technique by Chen *et al.* [112].
- Buggy Function combined with GitHub Issue, denoted as T_2, an approach similar to the technique by Fakhoury [109].
- Buggy Function alongside Error Information, denoted as T_3, an approach similar to the technique by Keller *et al.* [113].

For a fair comparison, we supply the same prompts, built from these facts, to MANIPLE and all the baselines. These prompts, unsurprisingly, differ structurally from the prompts on which the baselines were run. For example, our prompt incorporates our chain-of-thought instructions defined in supplementary materials (Section 6.10). Our focus here is on comparing MANIPLE's performance to the baselines' *w.r.t.* facts, not their performance given approach-specific optimal prompts, whose construction would require close cooperation with the authors of each baseline.

We evaluated tool efficacy according to two criteria: (1) number of plausible fixes (*i.e.* eq. (6.2)), and (2) the number of correct fixes, determined by the first plausible patch generated by the LLM and subsequently subjected to manual evaluation.

For assessing patch correctness, we employ an interrater agreement scale, where "3" indicates patches syntactically equivalent to the developer's patch, allowing for minor refactoring or restructuring, "2" — patches that achieve the intended outcome through an alternate method, "1" — diverging from the developer's patch, rendering correctness indeterminable, and "0" — incorrect patches (irrelevant, incomplete,

Tool	#fixed	Correct	% Correct
T_0	44	7	16%
T_1	37	5	14%
T_2	63	18	29%
T_3	66	31	47%
MANIPLE	88	37	42%

Figure 6.5: Comparison of tool performance on the BGP157PLY2 test set, focusing on bugs fixed. **#fixed** represents the number of bugs with test-passing patches, and **Correct** indicates bugs fixed with patches identical to the developer's. We observe that MANIPLE outperforms all the fact set combinations used. This shows that bug-tailored fact selection can improve the repair success.

introducing regressions). Each patch undergoes evaluation by two independent raters. In cases of scoring discrepancies, the raters discuss them. If the discrepancy is not resolved, the more conservative (lower) score is recorded, ensuring a rigorous standard of correctness. Following this scoring system, patches with the scores of 2 and 3 are labelled as "correct", and with the scores of 1 and 0 are labelled as "incorrect."

The analysis in Figure 6.5 underscores the potential of bug-tailored prompt selection for boosting automated program repair efficacy. Previous LLM-based zero-shot non-conversational approaches, represented by T_1, T_2 and T_3, that always use the same set of facts achieve a maximum of 66 plausibly fixed patches on BGP157PLY2. In contrast, MANIPLE, which leverages bug-tailored fact selection, identifies a significantly higher number of plausible patches (88). Furthermore, MANIPLE boasts 6 additional correct fixes compared to the best of these approaches. These findings suggest that tailoring the fact set to the specific context of each bug has the potential to improve the upper bound for repair success.

Surprisingly, T_1, which utilises function code alongside scope information (including class information and invoked functions), fixes fewer bugs compared to using function code alone. However, this does not imply that scope and class information are useless. T_1 still identifies 5 plausible patches, 2 of which are correct, that would not be found using just the function code as the only fact.

6.7 Threats to validity

In assessing GPT-3.5, we face an inherent challenge due to the possibility of data leakage. This risk stems from the fact that the datasets utilised for training these models are not openly available, obscuring the exact nature of the information they have been exposed to. As a result, there is a possibility that GPT-3.5 may have previously encountered instances from the test set during their training. The fact that our baselines use the same model partially addresses this threat.

The facts collected for our study represent our best effort. Although, to the best of our knowledge, this work considers the widest range of information in APR literature, we acknowledge that our fact set is provisional and will undoubtedly change as research into LLM-assisted APR continues.

The external validity of our findings relies on the distribution of bugs within our dataset. Our dataset is a subset of BugsInPy, a benchmark published at FSE'20 [64]. BugsInPy is a curated dataset built to best practice from GitHub repos with more than 10k stars that have at least one test case in the fixed commit that distinguishes the buggy version from its fix. Its representativeness has not been challenged and, from first principles, we can think of no reason that our filtering (Section 6.3.1) would introduce systematic bias. LLM performance on few-shot prompts has been shown to be sensitive to the order of the shots [133]; another threat to our external validity is that we did not permute facts across trials in our experiments.

6.8 Analyzing the impact of nondeterminism on LLM's performance

In this investigation, we aim to determine the influence of nondeterminism on the performance of Large Language Models (LLMs). The study is conducted using BGP32, as it is expensive to run a query a large amount of responses from the LLM. A challenge presented by nondeterminism in LLMs is the variability in outcomes from one experiment to the next, precluding the use of the conventional performance evaluation method of generating k responses from the LLM and determining success if at least one response meets the success criteria. However, due to the variability

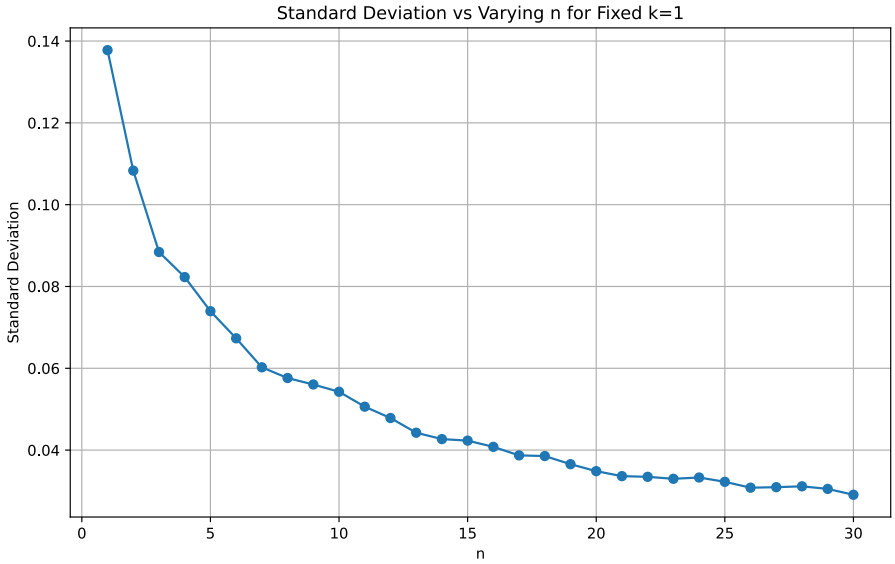


Figure 6.6: This plot illustrates the relationship between the standard deviation of pass@1 and the number of responses (n). The reduction in standard deviation demonstrates diminishing returns as n increases. This observation motivated our selection of $n = 15$, as its standard deviation is comparable to that of $n = 30$.

inherent in each trial, this approach may yield inconsistent results.

To address this variability, we use pass@k as our measure. Pass@k represents the probability of achieving at least one successful outcome within k attempts at solving a problem. This is determined by soliciting $n > k$ responses from the API and calculating the likelihood of at least one success within k trials, as discussed in Section 6.3.1.

For our methodology, we obtained $n = 30$ responses and evaluated pass@k for $k = 1$. To simulate multiple runs, we employed bootstrapping by sampling n responses from the pool of 30 responses with replacement, repeated 10 times. The standard deviation of pass@1 for these 10 samples was computed. Furthermore, we calculated the mean standard deviation of all the bugs in BGP32.

As depicted in Figure 6.6, the mean standard deviation of pass@1 demonstrates diminishing returns from approximately $n = 15$. This observation prompted our decision to choose $n = 15$. Figure 6.7 displays the mean standard deviation of pass@k across varying response counts (n) and trial counts (k). Notably, the heatmap

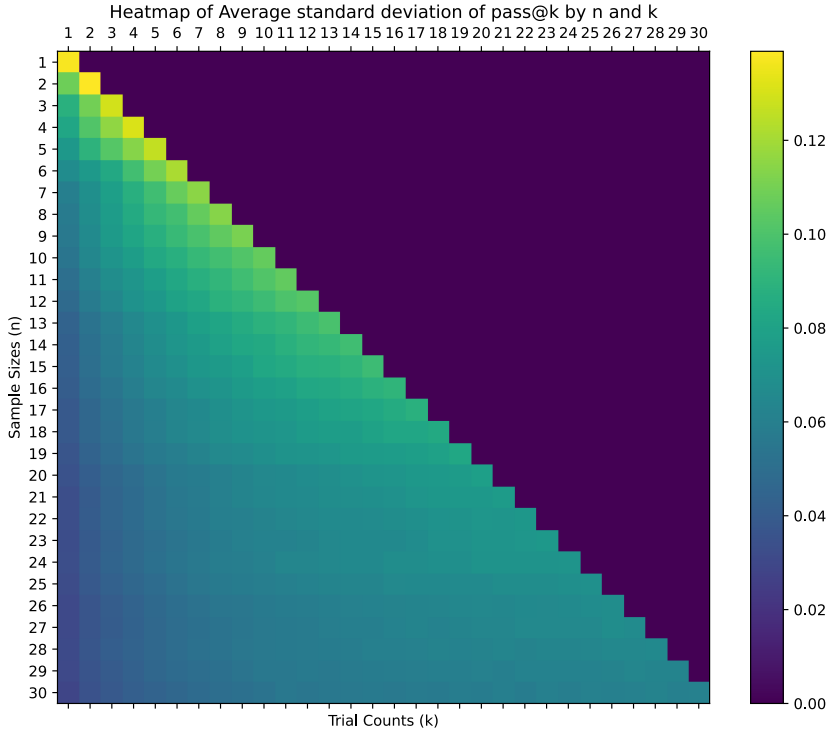


Figure 6.7: This heatmap depicts the standard deviation of pass@k against varying response count (n) and trial counts (k). The vertical gradient transitions from lighter to darker shades as n increases, signifying a reduction in standard deviation and thereby highlighting enhanced measurement precision with number of queries. On the horizontal axis, the gradient shifts from darker to lighter shades as the k grows, indicating an increase in standard deviation. This pattern suggests that lower values of k and higher values of n are associated with more precise outcomes.

indicates a decrease in standard deviation with increasing n, while an increase in k corresponds to higher standard deviation.

In understanding the mean standard deviation of pass@k, it is crucial to consider the granularity of the measure. Granularity refers to the smallest increment by which the measure can vary, encompassing all possible values within its range. Lower granularity allows for the capture of finer differences. However, if the mean standard deviation significantly exceeds the granularity, it can suggest that much of the finer differences in the output could be attributed to noise stemming from the nondeterminism of the LLM's output.

For our analysis, we specifically focus on $k = 1$ because the granularity for pass@1 is $\frac{1}{n}$ for n responses. This granularity allows for a more precise measurement of success probability since the standard deviation is lower than the granularity. However, for $k = 2$, the granularity for pass@2 is $\frac{1}{\binom{n}{2}}$, which is relatively small compared to any chosen n from our current data. For instance, pass@2 for $n = 15$ yields 0.05, which is larger than $\frac{1}{\binom{15}{2}} \approx 0.01$. Hence, we adopt $k = 1$ to ensure more accurate results.

6.9 Fact Prompt Templates

This section defines fact templates that are used to generate APR prompts.

Buggy Function (Directive) The body of the buggy function is given as the first section using the following template:

```
Please fix the buggy function provided below and output a corrected version.
Following these steps:
<CHAIN-OF-THOUGHT INSTRUCTIONS>

# The source code of the buggy function
```python
# this is the buggy function you need to fix
<FUNCTION BODY>
```
```

Buggy Class (1.1) The declaration of a class containing the buggy function is added to the function body section:

```
# The source code of the buggy function
```python
# The declaration of the class containing the buggy function
  class <CLASS DECLARATION>:
  ...

  # this is the buggy function you need to fix
  <FUNCTION BODY>
```
```

A class docstring offers insights into the overall purpose and functionality of the class, which can guide the LLM in understanding how the buggy function should operate. It is added to the buggy class declaration in the prompt using the standard

Python docstring notation.

Used Method Signatures (1.2) The methods used within the buggy function are incorporated into the buggy class declaration section:

```
The source code of the buggy function
```python
# The declaration of the class containing the buggy function
class <CLASS DECLARATION>:
  ...

  # This function from the same class is called by the buggy function
  def <FUNCTION SIGNATURE>:
    # Please ignore the body of this function

  ...
```
```

Signatures of the methods used, which are declared outside the class of the buggy function, are incorporated into the prompt as follows:

```
# Buggy function source code
```python
# This function from the same file, but not the same class, is called by the buggy
    ↪ function
def <FUNCTION SIGNATURE>:
    # Please ignore the body of this function

  ...
```
```

Failing test (1.3) The test code is incorporated into prompts in a separate section:

```
# A test function that the buggy function fails:
```python
# The relative path of the failing test file: <TEST FILE NAME>

<TEST CODE>
```
```

Error Information (2.1) We incorporate the error message and the stack trace into a separate section of the prompt:

```
# The error message from the failing test
```text
<ERROR MESSAGE>
```
```

```
<STACK TRACE>
```
```

**Runtime Information (2.2)** Assume that $x_1$, ..., $x_n$ are local variables in the buggy function, $v_1$, ..., $v_n$ and $t_1$, ..., $t_n$ are their values and types at the beginning of the function's execution, and $v'_1$, ..., $v'_n$ and $t'_1$, ..., $t'_n$ are their values and types at the end of the function's execution. To represent runtime values inside the prompt, we use the following format.

```
# Runtime values and types of variables inside the buggy function

Each case below includes input parameter values and types, and the values and
    ↪ types of relevant variables at the function's return, derived from
    ↪ executing failing tests. If an input parameter is not reflected in the
    ↪ output, it is assumed to remain unchanged. Note that some of these values
    ↪ at the function's return might be incorrect. Analyze these cases to
    ↪ identify why the tests are failing to effectively fix the bug.

# Case <CASE ID>
## Runtime values and types of the input parameters of the buggy function
x_1, value: v_1, type: t_1
...
x_n, value: v_n, type: t_n

## Runtime values and types of variables right before the buggy function's return
x_1, value: v'_1, type: t'_1
...
x_n, value: v'_n, type: t'_n

...
```

Angelic Values (2.3) Assume that the function operates a set of variables x_1, ..., x_n, for which the runtime values in the beginning of the function are v_1, ..., v_n, and the types are t_1, ..., t_n, and the angelic values at the end of the function are a_1, ..., a_n and the types are at_1, ..., at_n. Then, this information is incorporated into the prompt as follows:

```
# Expected values and types of variables during the failing test execution
Each case below includes input parameter values and types, and the expected values
    ↪  and types of relevant variables at the function's return. If an input
    ↪ parameter is not reflected in the output, it is assumed to remain unchanged
    ↪ . A corrected function must satisfy all these cases.
```
```

```
# Expected case <CASE ID>
# The values and types of buggy function's parameters
x_1, expected value: v_1, type: t_1
...
x_n, expected value: v_n, type: t_n

## Expected values and types of variables right before the buggy function's return
x_1, expected value: a_1, type: at_1
...
x_n, expected value: a_n, type: at_n


...
```

To reduce the length of the prompt, we only print the values of variables that change after the function execution. Values converted into a human-readable representation using Python's `.__str__()` method. Also, class and function variables are filtered.

GitHub Issue (3.1) We incorporate the GitHub issue's title and description into the prompt as follows:

```
# A GitHub issue for this bug
The issue's title:
'''text
<ISSUE TITLE>
'''
'''text
The issue's detailed description:
<ISSUE DESCRIPTION>
'''
```

6.10 Chain-of-thought Instructions

When writing these prompts, we also applied the standard techniques, chain-of-thoughts (CoT) prompting. Since we considered different possible subsets of facts for the inclusion in APR prompts, we used the following instruction template at the beginning of each prompt:

```
  1. Analyze the failing test case and its relationship with <LIST_OF_FACTS>.
  2. Identify the potential error location within the problematic
function.
  3. Explain the bug's cause using:
        <LIST_OF_FACTS>
```

```
  4. Suggest possible approaches for fixing the bug.
  5. Present the corrected code for the problematic function such that it
      ↪ satisfied the following:
        <CORRECTNESS_CRITERIA>
```

For bitvector = 1111111, the following will be its chain of thought instruction:

```
  1. Analyze the failing test case and its relationship with the error message
      ↪ along with the buggy function, buggy class, buggy file, the github issue,
      ↪  the expected and actual input/output variable information .
  2. Identify the potential error location within the problematic
function.
  3. Explain the bug's cause using:
        a. The buggy function
        b. The buggy class
        c. The buggy file
        d. The failing test and error message
        e. Discrepancies between expected and actual input/output variable values
        f. The Github Issue information
  4. Suggest possible approaches for fixing the bug.
  5. Present the corrected code for the problematic function such that it
      ↪ satisfied the following:
        a. Passes the failing test.
        b. Satisfies the expected input/output variable values provided.
        c. Successfully resolves the issue posted in Github
```

We conducted a small-scale experiment on BGP32 to confirm the effectiveness of CoT. The proportion of prompts that successfully led to a fix, incorporating CoT, stood at 0.46; this figure fell to 0.38 in the absence of CoT. Our findings indicated a modest positive Spearman correlation of 0.08 between the application of CoT and the pass@1 rate for repair success, which was supported by a statistically significant p-value of 10^{-13}. This led us to directly include Chain of Thought into our prompts.

6.11 Handling Imports in Prompts

Without listing import statements of the current file in APR prompts, the LLM may generate arbitrary import statements and call functions that are not defined within the repository. This behavior can be attributed to two main reasons. First, the LLM might lack contextual knowledge about the identifiers used within the buggy function. Second, it might assume the absence of import statements in the program and, based on its training data, add imports. Note that this is often not related to the functional

correctness of generated code, but incorrect imports make it hard to extract patches from the LLM's responses and insert them into the buggy programs.

For example, consider pandas:84. The LLM encounters a piece of code that uses the `BlockManager` identifier, which is an internal component of 'pandas'. Without specific context or import statements, the LLM might incorrectly suggest importing `BlockManager` directly from the top-level `pandas` package with:

```
from pandas import BlockManager
```

However, the appropriate way to import `BlockManager` is from within the `pandas.core.internals` module, which is more specific and not immediately apparent without domain knowledge or explicit instruction:

```
from pandas.core.internals import BlockManager
```

To test whether adding import statements improves pass@k, we manually selected 10 bugs which frequently resulted in undefined identifier errors and conducted two experiments for a comparative study. In the first experiment, we enumerated all 64 possible bitvectors, set the seed to 42, and the temperature to 1, and obtained the fix patches from the LLM. The pass@5 calculated in this experiment was 0.233. Then, we kept the settings the same and added the following instruction at the beginning of our prompt to obtain fix patches from the LLM:

````
Assume that the following list of imports is available in the current environment,
    ↪ so you do not need to import them when generating a fix.
```python
<import statements>
```
````

The pass@5 calculated from the above example was 0.271, which suggests a noticeable enhancement in the fix rate.

We did not consider import statements as a fact for our study, because it mostly solved a technical issue that helped us extract patches from the responses, rather than affecting the functional correctness of the generated code.

Chapter 7

Related Work

This section discusses the relevent areas that this thesis is related to, namely: symbolic execution, automated program repair, program analysis and prompt engineering.

7.1 Symbolic Execution

Symbolic execution [134, 135] is recognised as one of the most expensive testing methodologies [136]. Much research seeks to improve the efficiency and effectiveness of symbolic execution [88]. Variants of symbolic execution have also been proposed to reduce its cost, such as concolic execution [137, 138] and execution-based testing [56, 139]. Researchers have studied ways to accelerate path exploration in symbolic execution. One such approach is distributing path exploration among different workers [140, 141]. Several techniques have sought to leverage compositionality to speed up symbolic execution [142, 143]. Researchers also investigated techniques for pruning the search space [144, 145], and transforming the underlying code for symbolic execution [146, 147, 148].

One way to tackle symbolic execution's scalability challenge is to side-step it via search: rather than directly speed symbolic execution or solving, use search to maximise your computational budget. Kapus *et al.*'s Pending Constraints approach prioritises execution paths that are already known to be feasible and defer the rest [96]. Thus, like PSP, it avoids wasting resources on solving constraints. However, pending Constraints is more conservative: uses caching to identify *known-to-be-feasible*

paths; PSP, discussed in Chapter 5, in contrast, prioritises paths that it predicts will produce a likely state. Both approaches exemplify the "rich get richer" proverb: Pending Constraints will tend to explore feasible paths and their easy extensions more deeply, while PSP focuses on probable paths. For bug finding, we have shown PSP's prioritisation works better (Section 5.6.2).

Abstract symbolic execution (ASE) defines a value set decision procedure based on strided value interval sets for efficiently determining precise, or under-approximating value sets for variables, which helps to reduce the number of SMT queries. Furthermore, PSP approximates even further by reducing the number of queries as shown in Section 5.6.2.

Neuro-symbolic execution [149] trains a neural network to approximate hard-to-analyse program constructs such as loops and external function calls. In contrast, we directly approximate program states, which enables us to create an efficient search strategy reducing the number of SMT queries.

State Merging Although various general-purpose state-merging strategies [150, 151] for symbolic execution have been proposed, they have two important limitations in the context of assignment synthesis. First, to align symbolic memory in complex real-world programs, general-purpose state-merging strategies make assumptions about the topology of the states to merge. For example, KLEE [56] assumes that the merged states have exactly the same allocated memory objects and the same set of symbolic variables. These assumptions do not hold in the context of assignment synthesis, reducing the effectiveness of such techniques as demonstrated in Section 3.5. Second, a generated patch can modify several memory locations, and the number of such locations has to be bounded during path exploration to prevent path explosion. However, it is not clear how general-purpose state-merging techniques can efficiently express such a bound.

The key difference of general-purpose state-merging approaches from our techniques (Section 3.3.2) is that they operate on regions of code that impact of which on the state they seek to merge with a dedicated split and merge operations. However, path explosion can occur within these regions. Trident, which is discussed

in Chapter 3, builds state merging for the impact of patches in its search space into its very representation of symbolic state, obviating explicit split and merge operations. This representation incorporates cardinality constraints to bound the number of memory locations that can be modified by the synthesised patch. To compute function summaries, Trident uses symbolic execution with loop unrolling as in compositional symbolic execution [50].

7.2 Automated Program Repair

Test-overfitting in Program Repair: Test-overfitting is a central challenge of test-driven program repair [86]; it affects both generate-and-validate [128] and semantic [152] techniques. Researchers have proposed various approaches to tackle this problem. The first group of approaches uses a pre-defined database of transformations to increase the chance of generating correct patch [23, 24, 25]. The second group generates additional tests [10, 11]. The third group of techniques defines a cost function that assigns lower cost to patches that are more likely to be correct [26, 27, 28, 29, 3]. Trident, which is discussed in Chapter 3, complements existing techniques by proposing a new cost function specialized for the class of defects that require patches with side effects.

Program Repair Benchmarks in C: For evaluating Trident's side effects, we constructed new bug datasets, rather than directly using existing benchmarks, such as ManyBugs [51], IntroClass [51], Codeflaws [52], DBGBench [153]. We did not use ManyBugs, because the majority of its bugs either require side-effect-free patches or complex patches than involve many lines in several files. We did not use DBGBench, because it did not contain enough bugs with side-effect-free modifications, and some of its bugs were not reproducible with KLEE. IntroClass contains only very small programs. We reused some defect classes from Codeflaws that involve the insertion of assignments or functions calls. For evaluating the general performance of Trident and Rete, we employed 37 bugs taken from ManyBugs [51]. However, when running it on Rete, we dropped 2 bugs as we were unable to run Rete on these bugs due to the ML library dependencies.

Patch Generation and Checking Various patch generation approaches have been proposed: SPR [9] explores the space of patches by enumeration, GenProg [45] uses meta-heuristic search, CoCoNut [12] and Cure [13] use neural machine translation, and techniques like SemFix [8] and Angelix [4] employs SMT-based program synthesis. Program repair typically realises patch checking by 1) testing as in generate-and-validate techniques [45] or by 2) solving constraints as in semantic techniques [8, 102, 59]. Rete is a generic framework that does not impose a fixed patch generation or checking technique. In this work, we evaluate three instantiations of Rete: for the plastic surgery hypothesis [47], for Prophet's enumerative synthesiser [53], and for Trident's constraint-based synthesiser Section 3.3.3.

Patch Prioritisation Various patch prioritisation techniques have been proposed. DirectFix [26] prioritises smaller changes. Prophet [3] learns a probabilistic model for ranking patches. CapGen [154] uses AST node information to estimate the likelihood of concrete patches. GetaFix [5] uses a hierarchical clustering algorithm that clusters mined fix patterns into general and specific fix patterns and uses code context to choose an appropriate fix pattern. These approaches ignore information about program variables. Several recent techniques [61, 82], although they focus only on variables in the local context, do learn variable-related features and thus can prioritise variables. In contrast to these techniques, Rete does not require feature engineering and outperforms our language-agnostic feature engineering approach, as shown in Section 4.5.

Deep Learning for Program Repair Deep learning has been applied for automatically repairing bugs. DeepRepair [155] leverages learned code similarities using recursive autoencoders [156] to select repair ingredients from code fragments that are similar to the buggy code. Several techniques leverage deep learning to directly sort and transform code [157, 12]. CoCoNut [12] is a generate and validate approach that directly generates multiple patches by using an ensemble of context-aware neural network architecture. RewardRepair [158] uses execution data to improve upon patch synthesis. VarCLR [159] uses Recoder [160] synthesises a sequence of edits over directly synthesising the correct program. SequenceR [46] clusters

similar variables by directly passing a stream of tokens to an encoder. Although deep learning techniques implicitly learn information about the program namespace, our experiments show that state-of-the-art deep learning based tools have difficulty handling the long-range dependencies problem [62]. Rete addresses this problem by combining deep learning with program analysis in the form of extraction of CDU chains to learn a project-independent representation of the program namespace. We could not use Cure [13] as a baseline since its reproduction package is not currently public. Additionally, more recent tools, such as Recoder [160] and RewardRepair [158] were not compared against in the evaluation since they were implemented in Java.

Variable Representation There are various techniques used to represent variables, each with its own strengths and weaknesses. Word2vec [161] and its extensions, such as GloVe [162] and FastText [163], are simpler approaches that model variable representations by encoding tokens. However, these techniques may not capture the full complexity and nuance of natural language. To overcome these limitations, more advanced techniques, such as ELMo [164] and BERT [165], use pre-trained language models to achieve a deeper understanding of language [39, 166]. These approaches have been successfully applied in a range of code modification tasks, including suggesting variable names from code contexts [167], rewriting method and class names [168], and automated program repair [155, 61, 82]. They have also been used for type inference from natural language information [169, 170] and detecting bugs [171, 172]. These techniques, while possessing substantial strengths, are nonetheless impeded by sparse data, particularly when learning about global information outside the network's context window. This can result in suboptimal performance in certain code modification tasks. Rete ameliorates this data sparsity issue by using CDU chains, which contain important information that increases the likelihood of incorporating global information into the network's context. This approach is effective, as demonstrated by the results in Table 4.9, which show that Rete generates more fixes related to global variables when compared with other techniques.

Program repair with contextual information CoCoNut [12] utilizes surrounding contextual information to train an ensemble of neural machine translation models. Rete [99] employs Conditional Def-Use chains as context for CodeBert [78]. CapGen [154] utilises AST node information to estimate the likelihood of patches. DLFix [173] treats the program repair task as a code transformation task, learning to transform by additionally incorporating the surrounding context of the bug. The context used in this work includes the class and scope information of the buggy program, which is broader than the contexts used above. FitRepair [108] constructs prompts using identifier extracted from lines that look similar to the buggy line. Although, our work does not directly provide identifiers statically, as python is dynamic, we provide dynamic values of the variables during the test run.

LLM-based program repair LLM based techniques are making strides in APR. InferFix [22] uses few-shot prompting to repair issues from Infer static analyser. ChatRepair [110] uses interactive prompting constructed using failing test names and their corresponding failing assertions. Our approach focuses on the zero-shot, non-conversational setting, but can be potentially integrated with InferFix and ChatRepair. Various approaches focused on prompt engineering for APR, *e.g.* incorporating bug-related information within the prompts [22, 174, 175], as the quality of fixes could be enhanced by integrating contextual information, such as the bug's local context [111] and details about relevant identifiers [108], into the prompt. Xia *et al.* [108] utilize relevant identifiers to augment the fix rate of prompts. Keller *et al.* [113] reveals that, for debugging tasks, focusing on the specific line indicated by a stack trace is more effective than providing the entire trace. Similarly, Fakhoury *et al.* [109] investigates how combining issue descriptions with bug report titles and descriptions enhances program repair efforts. Following recent research [107], we refer to such pieces of information as *facts*. Our work uses individual facts from previous work to formulate and motivate the fact selection problem.

Program repair for Python QuixBugs [176] is a benchmark consisting of small programs in Java, and Python. They are not reflective of real software projects. Bugswarm [177], was constructed by automatically mining failing CI builds, and

thus contains issues outside of the scope of our study, such as configuration issues. BugsInPy [64] manually curates 501 bugs from 17 popular Python Projects. For Rete, which is discussed in Chapter 4, we choose 107 bugs in the defect class of Rete for evaluation, whereas for Maniple, which is discussed in Chapter 6, we selected 314 bugs from this benchmark that require modifications within a single function, and which we managed to reproduce. PyTER [178] is a program repair technique that focuses on Python TypeErrors; it was evaluated on a custom benchmark for type errors.

7.3 Program Analysis

Data-Driven Program Analysis Data-driven automatic tuning has been shown to improve static analysis performance [179, 180]. In contrast, we focus on the precision of program analysis. Heo *et al.* [181] use machine learning to balance the trade-off between precision and scalability of static analysis by selectively enabling unsoundness when analysing loops and library functions. PSP, which is discussed in Chapter 5, differs from these techniques by relying on data obtained through fuzzing. It directly approximates program states, enabling a wider range of applications, including symbolic execution and program repair.

Combining Testing and Verification Testing and verification have been combined in various ways [182]. A combination of concrete execution and abstraction to effectively reduce the under-approximation of abstraction has also been explored [183]. Tools such as SMASH [143] combine may and must analysis (over-approximation and under-approximation). UFO [184] uses interpolation to unify over and under-approximate techniques in model checking. PSP, which is discussed in Chapter 5, differs from these techniques by combining fuzzing data and abstract interpretation to construct probabilities for program states. This combination provides a trade-off between under-approximation and over-approximation and reduces false positives.

7.4 Prompt engineering

Recent advancements in prompt engineering have significantly influenced the effectiveness of models like ChatGPT. Notably, the tree-of-thoughts approach [185] and the zero-shot-CoT approach [186] have emerged as pivotal strategies. Frameworks like ReACT [187] use LLM to generate reasoning traces and task specific actions in an interleaved manner. Self Consistency [188] is an approach that traverses multiple diverse reasoning paths through few shot CoT and uses the generations to select the most consistent answer. Automatic Prompt Engineer [189] proposes a framework for automated prompt generation. It frames the task as a natural language synthesis task to construct prompts. This is orthogonal to our approach, as our task is to select ideal facts and the task of the Automatic Prompt Engineer is to refine the prompt into which the selected facts can be directly plugged. Repository Level Prompt Generation (RLPG) [190] is a very general framework for retrieving relevant repository context and constructing prompts, instantiated for code completion. RLPG generates prompt proposals and uses a classifier to choose the best one. In contrast, our fact selection problem discussed in Chapter 6 aims to find an optimal combination of facts to repair a given bug. Our work also uses a wider variety of information, incorporating, apart from code context, dynamic and external information.

Chapter 8

Conclusion

An effective Automated Program Repair (APR) tool must efficiently generate patches that developers are likely to accept. To achieve this, the tool must address three key challenges: efficient exploration of the search space of patches, mitigation of test overfitting, and generation of maintainable, human-like patches. This thesis presents four approaches aimed at tackling these challenges from different perspectives.

8.1 Contributions

In Chapter 3, Trident is presented as an efficient specification that makes navigating the search space more efficient. Patches generated by automated program repair (APR) tools usually require validation, and many APR tools go through a cycle of generating and validating patches until a successful one is found. Using the test suite as the validation criteria is time-consuming, as rebuilding the code with the generated patch and running the tests takes a considerable amount of time. Previous works such as Angelix [4], SemFix [8] and SE-ESOC [15] used concolic execution to construct a patch specification for validation. However, these approaches are unable to construct a patch specification for patches that cause side effects in the code. Trident provides an efficient method for constructing the patch specification while overcoming this limitation. Additionally, Trident proposes a patch priortisation strategy to alleviate test overfitting.

In Chapter 4, we address the issue of existing program repair approaches neglecting information about the program's namespace when searching for repairs.

This neglect reduces their efficiency and increases test overfitting, especially since the number of candidate variables is large. Rete aims to tackle this problem by enhancing patch prioritisation with information about the program namespace. This helps in effectively navigating through the large search space. By extracting information from CDU chains, Rete can prioritise candidate variables for patch generation more effectively. Our evaluation demonstrates that Rete can repair real bugs in open-source projects more rapidly compared to state-of-the-art methods while also finding more correct repairs.

In Chapter 5, we delve into the concept of Program State Probability (PSP), which is instrumental in enhancing program analyses such as abstract interpretation and symbolic execution. These analyses often suffer from imprecision caused by over- and under-approximation. PSP leverages execution samples to probabilistically estimate reachable program states, considering the probability of reaching a given state to enhance analysis precision. Additionally, it has been successfully applied in three domains: static analysis, symbolic execution, and program repair. The results demonstrate that using PSP improves the precision of abstract interpretation, increases the discovery of bugs through symbolic execution, and prioritises accurate patches for program repair. Section 5.4.3 shows that PSP can be leveraged to prioritise patches making APR techniques more efficient.

In Chapter 6, our research leverages the combination of program analysis and machine learning to create effective prompts for LLM-based automated program repair (APR). We use information obtained from program analysis to construct prompts that direct the LLM in generating correct patches. Some of these facts, such as Angelic Values capture the required execution semantics that the patch has to satisfy for the failing tests, making it easier for the LLM to generate a patch which satisfies the tests. Additionally, we tackle the challenge of selecting relevant facts, demonstrating that there is no universally optimal set of facts for addressing different types of bugs. Building on this understanding, we develop a bug-specific fact selection strategy to improve the efficacy of APR. Our research contributes to the efficient generation of human-like patches that are more likely

to be correct due to adapting the correct features, as demonstrated in Section 6.6. Additionally, this work presents important findings that will guide future researchers interested in building LLM-based agents for automatically resolving GitHub issues. It provides valuable insights into the utility of different types of facts, as discussed in Section 6.4.1, and highlights the non-monotonicity property, where adding more facts can actually decrease performance, as demonstrated in Figure 6.2. These findings help tool developers identify important facts based on different notions of utility and address the fact selection problem by choosing bug-specific facts, since the non-monotonicity property indicates that including too much information in the prompt can be detrimental.

8.2 Future Work

Some promising directions for future work are listed below:

Improving Variable prioritisation

Variable prioritisation is a crucial task, not only for Automated Program Repair (APR) but also for functionalities such as autocomplete for code. Chapter 4 uses CDU chains to effectively prioritise variables. I have explored additional strategies, such as utilising co-occurrence matrices and using compressed statement embeddings depicted as a graph, which outperformed CDU chains. Despite these advancements, there remains significant potential for further improvement. Therefore, continuing to explore and improving the representation is a promising area for future work.

Improving PSP

Chapter 5 employs basic approximations to estimate the probability of the feasibility of a program state. There is significant potential to enhance these approximation methods. PSP can be utilised to improve three distinct techniques: bug finding, Symbolic Execution, and Automated Program Repair. Each of these techniques can benefit from individually optimised algorithms that leverage PSP's insights. For instance, there is a wide range of strategies yet to be explored in utilising PSP's information to prioritise from the work list in choosing the next best state to explore during symbolic execution. Additionally, it is feasible to compress and utilise PSP's

data to create prompts for Large Language Models (LLMs), which may turn out to be an effective fact.

Exploring Additional Facts

We have only considered 16 facts (Figure 6.3) in Chapter 6. There are many additional facts which can potentially be considered to improve LLM based APR. Some examples include leveraging PSP, summaries of relevant functions, information such as class structures and interactions extracted from UML class and sequence diagrams.

Prompt Compression

Our observations in Chapter 6 highlight a challenge with Large Language Models (LLMs) — performance drops when the prompts are too long. Preliminary analysis of the dataset indicates a clear correlation between prompt length and pass@1 scores, as we have detailed in section 6.5.1. This insight has not only motivated the incorporation of prompt length as a critical feature in our random forest classifier but also sheds light on the non-monotonicity with respect to the number of facts, as adding more facts will increase the prompt length. Hence, the development of sophisticated prompt compression techniques emerges as a promising avenue for research.

Effective Fact Selection

Chapter 6 provides a proof of concept for a successful dynamic fact selection strategy by providing a simple statistical model which performs better than the best static fact selection strategy on our dataset. There is a substantial potential for further refinement. Potential future work could involve constructing a more robust dynamic fact selection strategy and comparing it against the established feature selection techniques used within the machine learning landscape.

Effective Prompt Construction

Chapter 6 addresses a strategy for selecting facts and describes the challenges in choosing effective facts. After selecting the required facts, a effective prompt is

essential for improving the performance of LLMs for any given task. While most current works focus on using fixed, manually crafted prompt templates, there is potential to automatically construct effective, bug-specific prompts. For automated program repair, this prompt construction task goes beyond traditional automated prompt construction on natural language tasks as it involves incorporating code fragments, execution values (from facts containing execution information), and other relevant facts. This can be accomplished by techniques such as fine-tuning LLMs, advanced search strategies, leveraging the framework of Generative Adversarial Networks [191], and leveraging optimisation algorithms. These methods can dynamically generate bug-tailored prompts, thereby enhancing the effectiveness of automated program repair.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[3] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312, 2016.

[4] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701, 2016.

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

[6] David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, and Federica Sarro. User-centric deployment of automated program repair at bloomberg. *arXiv preprint arXiv:2311.10516*, 2023.

[7] Casey Casalnuovo, Earl T Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. A theory of dual channel constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 25–28, 2020.

[8] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[9] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178, 2015.

[10] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841, 2017.

[11] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 8–18, 2019.

[12] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[13] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

[14] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[15] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. Symbolic execution with existential second-order constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 389–399, 2018.

[16] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra. Automatic program repair. *IEEE Software*, 38(04):22–27, jul 2021.

[17] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477, 2002.

[18] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[19] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019.

[20] Yi Li, Shaohua Wang, and Tien Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661–673. IEEE, 2021.

[21] Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. Program repair guided by datalog-defined static analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1216–1228, 2023.

[22] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*, 2023.

[23] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.

[24] Jindae Kim and Sunghun Kim. Automatic patch generation with context-based change application. *Empirical Software Engineering*, 24(6):4071–4106, 2019.

[25] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, pages 1–45, 2020.

[26] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, 2015.

[27] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*, pages 383–401. Springer, 2016.

[28] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604, 2017.

[29] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[30] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[31] Dexter C. Kozen. *Rice's Theorem*, pages 245–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 1977.

[32] João P Marques Silva and Karem A Sakallah. Grasp—a new search algorithm for satisfiability. In *The Best of ICCAD*, pages 73–89. Springer, 2003.

[33] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[34] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[35] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986.

[36] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[37] R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

[38] Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT press, 1999.

[39] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[40] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo:

Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[41] Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1229–1241, 2023.

[42] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[44] Sahil Verma and Subhajit Roy. Synergistic debug-repair of heap manipulations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 163–173, 2017.

[45] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

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

[47] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317, 2014.

[48] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd In-*

ternational Conference on Software Engineering, volume 1, pages 215–224. IEEE, 2010.

[49] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[50] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381. Springer, 2008.

[51] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

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

[53] Fan Long and Martin 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)*, pages 702–713. IEEE, 2016.

[54] David W Barron, John N Buxton, DF Hartley, E Nixon, and Christopher Strachey. The main features of cpl. *The Computer Journal*, 6(2):134–143, 1963.

[55] Clang: Llvm's default frontend. `https://clang.llvm.org/`. Accessed: 2021-03-03.

[56] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[57] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT workshop*, volume 2015, 2015.

[58] Docker OS virtualisation. `https://www.docker.com/`. Accessed: 2021-03-03.

[59] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering*, 2019.

[60] Martin Monperrus. A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242, 2014.

[61] Yingfei Xiong and Bo Wang. L2s: A framework for synthesizing the most probable program under a specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–45, 2022.

[62] Hai-Son Le, Alexandre Allauzen, and François Yvon. Measuring the influence of long range dependencies with neural network language models. In *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pages 1–10, 2012.

[63] Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart, and Pascal Hitzler. Neuro-symbolic artificial intelligence: Current trends, 2021.

[64] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al.

Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1556–1560, 2020.

[65] Łukasz Langa and collaborators. Black, a python code formatter. `https://github.com/psf/black`, 2022. Accessed: 2022-05-05.

[66] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[67] Keith Brian Gallagher. *Using program slicing in software maintenance*. PhD thesis, University of Maryland, Baltimore County, 1990.

[68] Tibor Gyimóthy, Arpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Software Engineering—ESEC/FSE'99*, pages 303–321. Springer, 1999.

[69] Syed Islam, Jens Krinke, David Binkley, and Mark Harman. Coherent clusters in source code. *Journal of systems and software*, 88:1–24, 2014.

[70] Eeshita Biswas, Mehmet Efruz Karabulut, Lori Pollock, and K Vijay-Shanker. Achieving reliable sentiment analysis in the software engineering domain using bert. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 162–173. IEEE, 2020.

[71] Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang. Sentiment analysis for software engineering: How far can pre-trained transformer models go? In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 70–80. IEEE, 2020.

[72] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[73] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2019.

[74] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pages 102–113. IEEE, 2019.

[75] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 615–627, 2020.

[76] John R Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, 1957.

[77] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[78] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama

Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[80] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[81] Dorit S Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In *Approximation algorithms for NP-hard problems*, pages 94–143. 1996.

[82] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 2021.

[83] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[84] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[85] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[86] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[87] K.J. Lieberherr and I.M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[88] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[89] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[90] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209, 2011.

[91] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[92] American fuzzy lop (afl) - a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed: May 5, 2023.

[93] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.

[94] Clam - static analyzer for llvm bitcode based on abstract interpretation. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed: May 5, 2023.

[95] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[96] Timotej Kapus, Frank Busse, and Cristian Cadar. Pending constraints in symbolic execution for better exploration and seeding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 115–126, 2020.

[97] Alireza S Abyaneh and Christoph M Kirsch. Ase: A value set decision procedure for symbolic execution. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 203–214. IEEE, 2021.

[98] Mythril: Security analysis tool for evm bytecode. `https://github.com/ConsenSys/mythril`. Accessed: May 5, 2023.

[99] Nikhil Parasaram, Earl T Barr, and Sergey Mechtaev. Rete: Learning namespace representation for program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1264–1276. IEEE, 2023.

[100] Parviz Moin. *Fundamentals of engineering numerical analysis*. Cambridge University Press, 2010.

[101] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541, 2020.

[102] Nikhil Parasaram, Earl T Barr, and Sergey Mechtaev. Trident: Controlling side effects in automated program repair. *IEEE Transactions on Software Engineering*, (01):1–1, 2021.

[103] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

[104] Jingxiu Yao and Martin Shepperd. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, pages 120–129, 2020.

[105] Introduction to smart contracts, 2022.

[106] Ernst Hellinger. Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. *Journal für die reine und angewandte Mathematik*, 1909(136):210–271, 1909.

[107] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr. Improving few-shot prompts with relevant static analysis products. *arXiv preprint arXiv:2304.06815*, 2023.

[108] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. Revisiting the plastic surgery hypothesis via large language models. *arXiv preprint arXiv:2303.10494*, 2023.

[109] Sarah Fakhoury, Saikat Chakraborty, Madan Musuvathi, and Shuvendu K Lahiri. Towards generating functionally correct code edits from natural language issue descriptions. *arXiv preprint arXiv:2304.03816*, 2023.

[110] Chunqiu Steven Xia and Lingming Zhang. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*, 2023.

[111] Julian Aron Prenner and Romain Robbes. Out of context: How important is local context in neural program repair? In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE, 2024.

[112] Yuxiao Chen, Jingzheng Wu, Xiang Ling, Changjiang Li, Zhiqing Rui, Tianyue Luo, and Yanjun Wu. When large language models confront repository-level automatic program repair: How well they done? *arXiv preprint arXiv:2403.00448*, 2024.

[113] Jan Keller and Jan Nowakowski. Ai-powered patching: the future of automated vulnerability fixes. Technical report, 2024.

[114] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 121–130, 2011.

[115] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.

[116] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR, 2023.

[117] The pandas:128 bug from bugsinpy. https://github.com/pandas-dev/pandas/commit/112e6b8d054f9adc1303138533ed6506975f94db, 2023. Accessed: 2024-03-21.

[118] John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, andCeron Felipe Juan Uribe, Liam Fedus, Luke Metz, Michael Pokorny, Rapha Gontijo Lopes, Shengjia Zhao, Arun Vijayvergiya, Eric Sigler, Adam Perelman, Chelsea Voss, Mike Heaton, Joel Parish, Dave Cummings, Rajeev Nayak, Valerie Balcom, David Schnurr, Tomer Kaftan, Chris Hallacy, Nicholas Turley, Noah Deutsch, Vik Goel, Jonathan Ward, Aris Konstantinidis, Wojciech Zaremba, Long Ouyang, Leonard Bogdonoff, Joshua Gross, David Medina, Sarah Yoo, Teddy Lee, Ryan Lowe, Dan Mossing, Joost Huizinga, Roger Jiang, Carroll Wainwright, Diogo Almeida, Steph Lin, Marvin Zhang, Kai Xiao, Katarina Slama, Steven Bills, Alex Gray, Jan Leike, Jakub Pachocki, Phil Tillet, Shantanu Jain, Greg Brockman, Nick Ryder, Alex Paino, Qiming Yuan, Clemens Winter, Ben Wang, Mo Bavarian, Igor Babuschkin, Szymon Sidor, Ingmar Kanitscheider, Mikhail Pavlov, Matthias Plappert, Nik Tezak, Heewoo Jun, William Zhuk, Vitchyr Pong, Lukasz Kaiser, Jerry Tworek, Andrew Carr, Lilian Weng, Sandhini Agarwal, Karl Cobbe, Vineet Kosaraju, Alethea Power, Stanislas Polu, Jesse Han, Raul Puri, Shawn Jain, Benjamin Chess, Christian Gibson, Oleg Boiko, Emy Parparita, Amin Tootoonchian, Kyle Kosic, and Christopher Hesse. Introducing chatgpt. *OpenAI blog*, Nov 2022.

[119] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[120] Lilian Weng. Prompt engineering. *lilianweng.github.io*, Mar 2023.

[121] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[122] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

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

[124] Pandas, python data analysis library. https://pandas.pydata.org/, 2023. Accessed: 2024-03-21.

[125] Matplotlib: Visualization with python. https://matplotlib.org/, 2023. Accessed: 2024-03-21.

[126] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. Llm is like a box of chocolates: the non-determinism of chatgpt in code generation. *arXiv preprint arXiv:2308.02828*, 2023.

[127] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[128] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy 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*, pages 532–543, 2015.

[129] Openai api. https://openai.com/blog/openai-api, 2024. Accessed: 2024-03-21.

[130] Eyal Winter. The shapley value. *Handbook of game theory with economic applications*, 3:2025–2054, 2002.

[131] Relative change. https://en.wikipedia.org/wiki/Relative_change, 2023. Accessed: 2024-03-21.

[132] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. Upset: visualization of intersecting sets. *IEEE transactions on visualization and computer graphics*, 20(12):1983–1992, 2014.

[133] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786*, 2021.

[134] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[135] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222, 1976.

[136] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071, 2011.

[137] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[138] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[139] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[140] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. *ACM Sigplan Notices*, 47(10):523–536, 2012.

[141] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 183–194, 2010.

[142] William R Bush, Jonathan D Pincus, and David J Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[143] Patrice Godefroid, Aditya V Nori, Sriram K Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 43–56, 2010.

[144] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. express: guided path exploration for efficient regression test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 1–11, 2011.

[145] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 144–154, 2012.

[146] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. Learning to accelerate symbolic execution via code transformation.

In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[147] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -overify: Optimizing programs for fast verification. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, number CONF. USENIX Association, 2013.

[148] Hayes Converse, Oswaldo Olivo, and Sarfraz Khurshid. Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 241–252. IEEE, 2017.

[149] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[150] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[151] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853, 2015.

[152] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, 2018.

[153] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 117–128, 2017.

[154] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE, 2018.

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

[156] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[157] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[158] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1506–1518. IEEE, 2022.

[159] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. Varclr: Variable semantic representation pre-training via contrastive learning. In *44th IEEE/ACM 44th International Conference on Software Engineering*, pages 2327–2339. ACM, 2022.

[160] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 341–353, 2021.

[161] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[162] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[163] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[164] ME Peters, M Neumann, M Iyyer, M Gardner, C Clark, K Lee, and L Zettlemoyer. Deep contextualized word representations. arxiv 2018. *arXiv preprint arXiv:1802.05365*, 12, 2018.

[165] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[166] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. Varclr: Variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2327–2339, 2022.

[167] Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

[168] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[169] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315. IEEE, 2019.

[170] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 607–618, 2016.

[171] Michael Pradel and Thomas R Gross. Detecting anomalies in the order of equally-typed method arguments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 232–242, 2011.

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

[173] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[174] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*, 2022.

[175] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020*, 2023.

[176] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, pages 55–56, 2017.

[177] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349. IEEE, 2019.

[178] Wonseok Oh and Hakjoo Oh. Pyter: effective program repair for python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 922–934, 2022.

[179] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[180] Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. A machine-learning algorithm with disjunctive model for data-driven program analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(2):1–41, 2019.

[181] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 519–529. IEEE, 2017.

[182] Dirk Beyer and Marie-Christine Jakobs. Cooperative verifier-based testing with coveritest. *International Journal on Software Tools for Technology Transfer*, 23:313–333, 2021.

[183] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: better together! In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 145–156, 2006.

[184] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In *Tools and Algorithms*

for the Construction and Analysis of Systems: 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings 18, pages 157–172. Springer, 2012.

[185] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

[186] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[187] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[188] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[189] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

[190] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR, 2023.

[191] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.