Automatic Inference of Code Transforms and Search Spaces for Automatic Patch Generation Systems
Fan Long, Peter Amidon, and Martin Rinard
Automatic Inference of Code Transforms and Search Spaces for Automatic Patch Generation Systems

Fan Long, Peter Amidon, and Martin Rinard
MIT EECS and MIT CSAIL
fanl@csail.mit.edu, peter@picnicpark.org, rinard@csail.mit.edu

Abstract
We present a new system, Genesis, that processes sets of human patches to automatically infer code transforms and search spaces for automatic patch generation. We present results that characterize the effectiveness of the Genesis inference algorithms and the resulting complete Genesis patch generation system working with real-world patches and errors collected from top 1000 github Java software development projects. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from successful patches.

1. Introduction
Automatic patch generation systems [15, 20–23, 27, 35, 42, 45, 46] hold out the promise of significantly reducing the human effort required to diagnose, debug, and fix software errors. The standard generate and validate approach starts with a set of test cases, at least one of which exposes the error. It deploys a set of transforms to generate a search space of candidate patches, then runs the resulting patched programs on the test cases to find plausible patches that produce correct outputs for all test cases.

All previous generate and validate systems work with a set of manually crafted transforms [20–23, 35, 42, 45, 46]. This approach limits the system to fixing only those bugs that fall within the scope of the transforms that the developers of the patch generation system decided to provide. This limitation is especially unfortunate given the widespread availability (in open-source software repositories) of patches developed by many different human developers. Together, these patches embody a rich variety of different patching strategies developed by a wide range of human developers, and not just the patch generation strategies encoded in a set of manually crafted transforms from the developers of the patch generation system.

1.1 Genesis
We present Genesis, a novel system that automatically infers transforms and resulting search spaces for automatic patch generation systems. Given a set of successful human patches drawn from available revision histories, Genesis automatically generalizes subsets of patches to infer transforms that together generate a productive search space of candidate patches. Genesis can therefore leverage the combined patch generation expertise of a many different developers to capture a wide range of productive patch generation strategies. It can then automatically apply the resulting transforms to successfully correct errors in multiple previously unseen applications. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from successful patches.

Transforms: Each Genesis transform has two template abstract syntax trees (ASTs). One template AST matches code in the original program. The other template AST specifies the replacement code for the generated patch. Template ASTs contain template variables, which match subtrees or subforests in the original or patched code. Template variables enable the transforms to abstract away patch- or application-specific details to capture common patch patterns implemented by multiple patches drawn from different applications.

Generators: Many useful patches do not simply rearrange existing code and logic; they also introduce new code and logic. Genesis transforms therefore implement partial pattern matching in which the template AST for the patch contains free template variables that are not matched in the original code. Each of the free template variables is associated with a generator, which systematically generates new candidate code components for the free variable. This new technique, which enables Genesis to synthesize new code and logic in the candidate patches, is essential to enabling Genesis to generate correct patches.

Search Space Inference with ILP: A key challenge in patch search space design is navigating an inherent trade-off between coverage and tractability [24]. On one hand, the search space needs to be large enough to contain correct patches for the target class of errors (coverage). On the other hand, the search space needs to be small enough so that the patch generation system can efficiently explore the space to find the correct patches (tractability) [24].

Genesis navigates this tradeoff by formulating an integer linear program (ILP) whose solution maximizes the number of training patches covered by the inferred search space while acceptably bounding the number of candidate patches that the search space can generate (Section 3.5).

The ILP operates over a collection of subsets of patches drawn from a set of training patches. Each subset generalizes to a Genesis transform, with the final search space generated by the set of transforms that the solution to the ILP selects. Genesis uses a sampling algorithm to tractably derive the collection of subsets of patches for the ILP. This sampling algorithm incrementally builds up larger subsets of patches from smaller subsets, using a fitness function to identify promising candidate subsets (Section 3.4). Together, the sampling algorithm and final ILP formulation of the search space selection problem enable Genesis to scalably infer a set of transforms with both good coverage and good tractability.

1.2 Experimental Results
We use Genesis to infer patch search spaces and generate patches for two classes of errors: null pointer errors (NPE) and out of bounds errors (OOB). The NPE patch training set includes 483 patches from 126 different applications; the OOB patch training set includes 199 patches from 117 different applications. For our benchmark set of 20 null pointer errors and 13 out of bounds errors, Genesis generates correct patches for 12 null pointer errors and 6 out of bounds errors. The time required for Genesis to generate the first correct patch for a given error is often less than a minute. All of the applications are large, real-world applications from github [4] or the MUSE corpus [7] with up to 235K lines of code. These results highlight the effectiveness of the Genesis
inference algorithms in finding productive patch search spaces for errors that occur in practice.

1.3 Contributions

This paper makes the following contributions:

• **Transforms with Template ASTs and Generators:** We present novel transforms with template ASTs and generators for free template variables. These transforms enable Genesis to abstract away patch- and application-specific details to capture common patch patterns and strategies implemented by multiple patches drawn from different applications. Generators enable Genesis to synthesize the new code and logic required to obtain correct patches for errors that occur in large real-world applications.

• **Patch Generalization:** We present a novel patch generalization algorithm that, given a set of patches, automatically derives a transform that captures the common patch generation pattern present in the patches. This transform can generate all of the given patches as well as other patches with the same pattern in the same or other applications.

• **Search Space Inference:** We present a novel search space inference algorithm. Starting with a set of training patches, this algorithm infers a collection of transforms that together generate a search space of candidate patches with good coverage and tractability. The inference algorithm includes a novel sampling algorithm that identifies promising subsets of training patches to generalize and an ILP-based solution to the final search space selection problem.

• **Complete System and Experimental Results:** We present a complete patch generation system, including error localization and candidate patch evaluation algorithms, that uses the inferred search spaces to automatically patch errors in large real-world applications. We also present experimental results from this complete system.

Automatic patch generation systems have great potential for automatically eliminating errors in large software systems. By inferring transforms and search spaces from sets of previous successful patches, Genesis can automatically derive patch generation strategies that leverage the combined insight and expertise of developers worldwide. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from previous successful patches.

2. Example

We next present a motivating example of using Genesis to infer a search space to generate a correct patch for a real world null-pointer exception (NPE) error (shown at the bottom of Figure 1).

**Collect and Split Training Set:** Genesis works with a training set of successful human NPE patches to infer a search space for repairing NPE errors. In our example, the training set consists of 483 human patches for NPE errors collected from 126 github repositories.

To avoid overfitting, Genesis reserves 121 (25%) human patches from the training set as a validation set (Section 3.4). This leaves 362 human patches remaining in the training set.

**Generalizing Patches:** The Genesis inference algorithm works with selected subsets of patches from the training set. For each subset, it applies a generalization algorithm to obtain a transform that it can apply to generate candidate patches. Figure 1 presents one of the selected subsets of patches in our example: the first patch disjoins the clause `mapperTypeElement != null` to an if condition, the second patch conjoins the clause `Material.getMaterials(getTypeId()) != null` to an if condition. These patches are from three different applications, specifically mapstruct [6] revision 6d7a4d, modelmapper [8] revision d85131, and Bukkit [2] revision f3115. Given these three patches, the Genesis generalization algorithm produces the transform in Figure 1. When applied, this transform can generate all of the three patches in the selected subset as well as other patches for other applications.

Each transform has an initial template abstract syntax tree (AST) and a transformed template AST. These template ASTs capture the syntactic contexts of the original and patched code, respectively. In our example, the initial template AST $T_0$ matches a boolean expression $A$ that occurs within a function body (if all of the patches had modified if conditions, the initial patch would have reflected that more specific context). The transformed template AST replaces the matched boolean expression $A$ with a patch of the form $A \ op_1 \ (B \ op_2 \ null)$, where $op_1 = C \in \{!,:,==\} \ and \ op_2 = D \in \{<,=\}$. $A$ is the original matched boolean expression, and $B$ is an expression produced by a generator. In this example, Genesis infers the generator that generates all expressions that satisfy the constraints in Figure 1, specifically, that $B$ is either a call expression or a variable, that the number of variables in $B$ is at most 2, that the number of calls in $B$ is at most 1, and that any variables or calls in $B$ must also appear in the original unpatched code. These constraints generalize the original three patches to appropriately scope the space of patches that the transform generates.

**Applying the Transform:** Figure 1 shows how Genesis applies the transform to patch a null pointer exception in another application, specifically in DataflowJavaSDK [3] revision c06125. Here the patch instantiates $B$ as the variable `unions`, $C$ as `==` and $D$ as `||` to disjoin the clause `unions == null` to the original if condition. The patch causes the enclosing function `innerGetOnly()` to return a predefined default value when `unions` is `null` (instead of incorrectly throwing a null pointer exception). Genesis uses fault localization techniques (Section 4) to select this if condition as a patch candidate.

The transform also generates 13 other candidate patches at this if condition. Genesis uses the DataflowJavaSDK JUnit [5] test suite (which includes 830 test cases) to filter out these other candidate patches (as well as other candidate patches from other transforms and other patch candidate locations in DataflowJavaSDK). For Genesis to successfully patch the exception, the test suite must contain an input that exposes the exception, i.e., that causes the application to throw the null pointer exception. Genesis finds plausible patches, i.e., patches that produce the correct output for all test cases, by running the patched application on all of the test cases (including the test case that exposed the exception) and checking the testing results. In the example, the patch in Figure 1 is the only plausible patch. This plausible patch is also correct and matches the subsequent human developer patch for this exception.

**Other Transforms:** Figure 2 summarizes three other null pointer exception transforms that Genesis infers (out of a total of 14 inferred null pointer transforms). The first transform patches method call expressions on null objects. The transform adds a guard expression of the form $A \ == \ null \ ? \ C \ : \ D$ that first checks if $A$ is `null` and, if so, evaluates to a constant such as `null` or 0 instead of throwing NPE. Note that the transform creates a new variable $D$ instead of reusing the original code — the human patches in the training set often slightly refactor the code instead of directly using the original code as the else expression.
The second transform in Figure 2 adds an if statement that executes an original statement \( A \) only if \( B \neq \text{null} \). This transform eliminates null pointer exceptions by simply skipping statements when they would otherwise throw null pointer exceptions. The third transform executes a return or continue statement \( C \) instead of an original statement \( A \) if \( B == \text{null} \). This transform eliminates null pointer exceptions by returning from the enclosing function or skipping the current loop iteration if subsequent code would throw a null pointer exception. All of these transforms capture null pointer exception patch patterns that appear in our training set of human null pointer exception patches.

**Search Space:** The Genesis candidate patch search space is determined by the set of inferred transforms that Genesis selects to generate candidate patches (operating in tandem with the error localization algorithm, see Section 4). To obtain an effective search space, Genesis must navigate a tradeoff between coverage (how many correct patches it can generate) and tractability (how long it takes to generate the search space, how long it takes to run the patched versions against the test suite, and how many plausible but incorrect patches it generates \([24]\)). Increasing the number of selected transforms tends to improve coverage but degrades tractability; decreasing the number of selected transforms tends to have the opposite effect.

Genesis includes a search space selection algorithm that uses integer linear programming to navigate this tradeoff (Section 3.5). The search space is determined by a collection of selected transforms generalized from subsets of training patches. The integer linear program selects a collection of transforms that 1) maximizes the number of validation patches in the collected validation set that are inside the search space subject to 2) the number of candidate patches that all selected transforms generate when applied to the original code for any of such validation patches is less than a chosen bound. The integer linear program therefore maximizes coverage while excluding transforms that generate unacceptably large search spaces.

It turns out that, in practice, essentially all useful transforms can be generated by generalizing relatively small subsets of training patches (six or fewer training patches). For small numbers of training patches, it might be feasible to simply generate all subsets of training patches smaller than a given bound, then use the integer linear program to select the search space. Because we work with too many training patches for this approach to scale, Genesis first uses a sampling algorithm that incrementally builds up promising subsets (starting with subsets of size two, then working up to subsets of size six). It uses a fitness function to prune unpromising subsets (Section 3.4). Once it has obtained a collection of promising subsets of training patches, it applies the integer linear program to obtain the final selected transforms.

### 3. Inference System

Given a set of training pairs \( D \), each of which corresponds to a program before a change and a program after a change, Genesis infers a set of transforms \( P \) which generates the search space.

Genesis obtains the search space in two steps: 1) it first runs a sampling algorithm to obtain a set of candidate transforms, each of which is generalized from a subset of changes in \( D \) and 2) it then selects a subset of the candidate transforms, formulating the trade-off between the coverage and the tractability of the search space as an integer linear programming (ILP) problem. It invokes an off-the-shelf ILP solver \([16]\) to select the final set of transforms.

Sections 3.1 and 3.2 present definitions and notation. Section 3.3 presents definitions for the generalization function which derives candidate transforms from a set of program changes. Section 3.4 presents the sampling algorithm. Section 3.5 presents the search space inference algorithm. We discuss Java implementation details in Section 3.6.
3.1 Preliminaries

The Genesis inference algorithm works with abstract syntax trees (ASTs) of programs. In this section, we model the programming language that Genesis works with as a context-free grammar (CFGs) and we model ASTs as the parse trees for the CFG. Note that although the current implementation of Genesis is for Java, it is straightforward to extend the Genesis inference algorithm to other programming languages as well.

Definition 1 (CFG). A context-free grammar (CFG) $G$ is a tuple $(N, \Sigma, R, s)$ where $N$ is the set of non-terminals, $\Sigma$ is the set of terminals, $R$ is a set of production rules of the form $a \rightarrow b_1b_2\ldots b_n$ where $a \in N$ and $b_i \in N \cup \Sigma$, and $s \in N$ is the starting non-terminal of the grammar. The language of $G$ is the set of strings derivable from the start non-terminal: $L(G) = \{ w \in \Sigma^* \mid s \Rightarrow^* w \}$.

Definition 2 (AST). An abstract syntax tree (AST) $T$ is a tuple $(G, X, r, \xi, \sigma)$ where $G = (N, \Sigma, R, s)$ is a CFG, $X$ is a finite set of nodes in the tree, $r \in X$ is the root node of the tree, $\xi : X \rightarrow \mathbb{R}^+$ maps each node to the list of its child nodes, and $\sigma : X \rightarrow (N \cup \Sigma)$ attaches a non-terminal or terminal label to each node in the tree.

Definition 3 (AST Traversal and Valid AST). Given an AST $T = (G, X, r, \xi, \sigma)$ where $G = (N, \Sigma, R, s)$, $\text{str}(T) = \text{traverse}(r) = \varnothing^* \cup \{ \perp \}$ is the terminal list obtained via traversing $T$ where

$$\text{traverse}(x) = \begin{cases} \text{traverse}(x_{c_1}) \cdots \text{traverse}(x_{c_k}) & \text{if } \sigma(x) \in N, \xi(x) = \{x_{c_1}, \ldots, x_{c_k}\}, \text{ and } x \rightarrow x_{c_1} \cdots x_{c_k} \in R \\ \sigma(x) & \text{if } \sigma(x) \in \Sigma \\ \perp & \text{otherwise.} \end{cases}$$

If the obtained string via traversal belongs to the language of $G$, i.e., $\text{str}(T) \in L(G)$, then the AST is valid.

We next define AST forests and AST slices, which we will use in this section for describing our inference algorithm. An AST forest is similar to an AST except it contains multiple trees and a list of root nodes. An AST slice is a special forest inside a large AST which corresponds to a list of adjacent siblings.

Definition 4 (AST Forest). An AST forest $T$ is a tuple $(G, X, L, \xi, \sigma)$ where $G$ is a CFG, $X$ is the set of nodes in the forest, $L = (x_1, x_2, \ldots, x_k)$ is the list of root nodes of trees in the forest, $\xi$ maps each node to the list of its children nodes, and $\sigma$ maps each node in $X$ to a non-terminal or terminal label.

Definition 5 (AST Slice). An AST slice $S$ is a pair $(T, L) = (G, X, r, \xi, \sigma)$, where $L = \langle r \rangle$ is a list that contains only the root node or $L = \langle x_1, \ldots, x_k \rangle$ is a list of AST sibling nodes in $T$ such that $\exists x' \in X : \xi(x') = \{x_1, \ldots, x_1, \ldots, x_k\}$ (i.e., $L$ is a sublist of $\xi(x')$).

Given two ASTs $T$ and $T'$, where $T$ is the AST before the change and $T'$ is the AST after the change, Genesis computes AST difference between $T$ and $T'$ to produce an AST slice pair $(S, S')$ such that $S$ and $S'$ point to the sub-forests in $T$ and $T'$ that subsume the change. For brevity, in this section we assume $D = \{ (S_1, S'_1), (S_2, S'_2), \ldots, (S_m, S'_m) \}$ is a set of AST slice pairs, i.e., Genesis already converted AST pairs of changes to AST slices.

Notation and Utility Functions: We next introduce notation and utility functions that we are going to use in the rest of this section. For a map $M$, $\text{dom}(M)$ denotes the domain of $M$. $M[a \mapsto b]$ denotes the new map which maps $a$ to $b$ and maps other elements in $\text{dom}(M)$ to the same values as $M$. $\emptyset$ denotes an empty set or an empty map.

$\text{nodes}(\xi, L)$ denotes the set of nodes in a forest, where $\xi$ maps each node to a list of its children and $L$ is the list of the root nodes of the trees in the forest.

$$\text{nodes}(\xi, L) = \bigcup_{i=1}^{k} \{ (x_1) \cup \text{nodes}(\xi, (x_i)) \}$$

where $L = (x_1, \ldots, x_k)$

$\text{nonterm}(L, X, \xi, \sigma, N)$ denotes the set of non-terminals inside a forest, where $L$ is the root nodes in the forest, $X$ is a finite set of nodes, $\xi$ maps each node to a list of children nodes, $\sigma$ attaches each node to a terminal or non-terminal label, and $N$ is the set of non-terminals:

$$\text{nonterm}(L, X, \xi, \sigma, N) = \bigcup_{i=1}^{k} \{ (\{ \sigma(x) \mid \sigma(x) \in N \}) \cup \text{nonterm}(\xi(x), X, \xi, \sigma, N) \}$$

where $L = (x_1, \ldots, x_k)$

$\text{inside}(S)$ denotes the set of non-terminals of the ancestor nodes of an AST slice $S$:

$$\text{inside}(S) = \{ \sigma(x) \mid \sigma(x') \in N \} \cup \text{inside}(T, \langle x' \rangle)$$

where $S = (T, L) = (G, X, r, \xi, \sigma)$, $G = (N, \Sigma, R, s)$

$L = (x_1, \ldots, x_k)$, and $\forall i \in [1, k] : x_i \in \xi(x')$

$\text{diff}(A, B)$ denotes the number of different terminals in leaf nodes between two ASTs, AST slices, or AST forests. If $A$ and $B$ differs in not just terminals in leaf nodes, $\text{diff}(A, B) = \infty$. $A \equiv B$ denotes that $A$ and $B$ are equivalent, i.e., $\text{diff}(A, B) = 0$. Figure 3 presents the detailed definitions of $\text{diff}(\cdot)$ and “$\equiv$”.

3.2 Template AST Forest, Generator, and Transforms

Template AST Forest: We next introduce the template AST forest, which can represent a set of concrete AST forest or AST slice. The key difference between template AST forest and concrete AST forest is that template AST forest contains template variables, each of which can match against any appropriate AST subtrees or AST sub-forests.

Definition 6 (Template AST Forest). A template AST forest $T$ is a tuple $(G, V, \gamma, X, L, \xi, \sigma)$, where $G = (N, \Sigma, R, s)$ is a CFG, $V$ is a finite set of template variables, $\gamma : V \rightarrow \{0, 1\} \times \text{Powerset}(N)$ is a map that assigns each template variable to a bit of zero or
one and a set of non-terminals, $X$ is a finite set of nodes in the subtree, $L = \langle x_1, x_2, \ldots, x_k \rangle$, $x_i \in X$ is the list of root nodes of the trees in the forest, $\xi : X \rightarrow X'$ maps each node to the list of its children nodes, and $\sigma : X \rightarrow N \cup \Sigma \cup V$ attaches a non-terminal, a terminal, or a template variable to each node in the tree.

For each template variable $v \in V$, $\gamma(v) = \langle b, W \rangle$ determines the kind of AST subtrees or sub-forests which the variable can match against. If $b = 0$, $v$ can match against only AST subtrees not sub-forests. If $b = 1$, then $v$ can match against both subtrees and sub-forests. Additionally, $v$ can match against an AST subtree or sub-forest only if its root nodes do not correspond to any non-terminal outside $W$.

Intuitively, each non-terminal in the CFG of a programming language typically corresponds to one kind of syntactic unit in programs at certain granularity. Template AST forests with template variables enable Genesis to achieve desirable abstraction over concrete AST trees during the inference. They also enable Genesis to abstract away program-specific details so that Genesis can infer useful transforms from changes across different programs and different applications.

**Definition 7** ("\(\Rightarrow\)" and "\(\Rightarrow_{slice}\)" Operators for Template AST Forest). Figure 4 presents the formal definition of the operators "\(\Rightarrow\)" for a template AST forest $T = \langle G, V, \gamma, X, L, \xi, \sigma \rangle$. "\(T \Rightarrow \langle T, M, W \rangle\)" denotes that $T$ matches the concrete AST forest $T$ with the template variable bindings specified in $M$, where $M$ is a map that assigns each template variable in $V$ to an AST forest. Figure 4 also presents the formal definition of the operator "\(\Rightarrow_{slice}\)". Similarly, "\(T \Rightarrow_{slice} (S, M)\)" denotes that $T$ matches the concrete AST slice $S$ with the variable bindings specified in $M$.

The first rule in Figure 4 corresponds to the simple case of a single terminal node. The second and the third rules correspond to the cases of a single non-terminal node or a list of nodes, respectively. The two rules recursively match the children nodes and each individual node in the list.

The fourth and the fifth rules correspond to the case of a single template variable node in the template AST forest. The fourth rule matches the template variable against a forest, while the fifth rule matches the template variable against a tree. These two rules check that the corresponding forest or tree of the variable in the binding map $M$ is equivalent to the forest or tree that the rules are matching against.

**Generators:** Many productive patches do not just rearrange existing components and/or logics in the changed slice, but also introduce useful new components and/or logic. We next introduce generators, which enable Genesis to synthesize such patches.

**Definition 8** (Generator). A generator $G = \langle G, b, \delta, W \rangle$, where $G = \langle N, \Sigma, R, s \rangle$ is a CFG, $b \in \{0, 1\}$ indicates the behavior of the generator, $\delta$ is an integer bound for the number of tree nodes, $W \subseteq N$ is the set of allowed non-terminals during generation.

Currently, generators in Genesis exhibit two kinds of behaviors. If $b = 0$, the generator generates a sub-forest with less than $\delta$ nodes that contains only non-terminals inside the set $W$. If $b = 1$, such a generator copies an existing sub-forest from the original AST tree with non-terminal labels in $W$ and then replaces up to $\delta$ leaf non-terminal nodes in the copied sub-forest.

**Definition 9** (Generation Operator "\(\implies\)" for Generators). Figure 5 presents the formal definition of the operator "\(\implies\)" for a generator $G$. Given $G$ and an AST slice $S = \langle T, L \rangle$ as the context, $\langle G, S \rangle \implies T'$ denotes that the generator $G$ generates the AST forest $T'$.

The first rule in Figure 5 handles the case where $b = 0$. The rule checks that the number of nodes in the result forest is within the bound $\delta$ and the set of non-terminals in the forest is a subset of $W$. The second rule handles the case where $b = 1$. The rule checks that the difference result forest and an existing forest in the original AST is within the bound and the root labels are in $W$.

Note that, theoretically, generators may generate an infinite number of different AST forests for programming languages like Java, because the set of terminals (e.g., identifiers and constants) is infinite. Genesis, in practice, places additional Java-specific constraints on generators to make the generated set finite and tractable (See Section 3.6).

**Transforms:** Finally, we introduce transforms, which generate the search space inferred by Genesis. Given an AST slice, a transform generates new AST trees.
\[ S = \langle \langle G, X, r, \xi, \sigma \rangle, L \rangle \quad A \subseteq \text{inside}(S) \]
\[
T_0 \models (S, M) \quad B = \{v_1 \mapsto G_1, v_2 \mapsto G_2, \ldots, v_m \mapsto G_m\}
\]
\[
\forall m \\iota=1 (\langle G_i, S \rangle = \Rightarrow T''_{\iota}) \quad M' = \{v_1 \mapsto T'_1, v_2 \mapsto T'_2, \ldots, v_m \mapsto T'_m\}
\]
\[
T_1 \models (T', M \cup M') \quad \langle S, T' \rangle \triangleright T \quad \text{str}(T) \in \mathcal{L}(G)
\]
\[
(A, T_0, T_1, B, S) \models T
\]

**Figure 6.** Definition of the operators “\(\Rightarrow\)” and “\(\Rightarrow_{\text{slice}}\)” for the transform \(P\)

**Definition 10** (Transform). A transform \(P\) is a tuple \((A, T_0, T_1, B)\). \(A : \text{Powerset}(N)\) is a set of non-terminals to denote the context where the transform can apply; \(T_0 = \langle G, V_0, \gamma_0, X_0, L_0, \xi_0, \sigma_0 \rangle\) is the template AST forest before applying the transform; \(T_1 = \langle G, V_1, \gamma_1, X_1, L_1, \xi_1, \sigma_1 \rangle\) is the template AST forest after applying the transform; \(B\) maps each template variable \(v\) that only appears in \(T_1\) to a generator (i.e., \(\forall v \in V_1 \setminus V_0, B(v)\) is a generator).

**Definition 11** (“\(\Rightarrow\)” and “\(\Rightarrow_{\text{slice}}\)” Operators for Transforms). Figure 6 presents the formal definition of the “\(\Rightarrow\)” and “\(\Rightarrow_{\text{slice}}\)” operator for a transform \(P\). \(\langle P, S \rangle \Rightarrow T'\) denotes that applying \(P\) to the AST slice \(S\) generates the new AST \(T'\). \(\langle P, S \rangle \Rightarrow_{\text{slice}} S'\) denotes that applying \(P\) to the AST slice \(S\) generates the AST of the slice \(S'\).

Intuitively, in Figure 6 \(A\) and \(T_0\) determine the context where the transform \(P\) can apply. \(P\) can apply to an AST slice \(S\) only if the ancestors of \(S\) have all non-terminal labels in \(A\) and \(T_0\) can match against \(S\) with a variable binding map \(M\), \(T_1\) and \(B\) then determine the transformed AST tree. \(T_1\) specifies the new arrangement of various components and \(B\) specifies the generators to generate AST sub-forests to replace free template variables in \(T_1\). Note that \(\langle S, T' \rangle \triangleright T\) denotes that the obtained AST tree of replacing the AST slice \(S\) with the AST forest \(T'\) is equivalent to \(T\).

### 3.3 Transform Generalization

The generalization operation for transforms takes a set of AST slice pairs \(D\) as the input and produces a set of transforms, each of which can at least generate the corresponding changes of the pairs in \(D\). We first present the generalization operator for generators then we present the generalization operator for transforms.

**Definition 12** (Generator Generalization). Figure 7 presents the definition of generalization function \(\psi(D)\). Given a set of of AST slice pairs \(D = \{(S_1, S'_1), (S_2, S'_2), \ldots, (S_m, S'_m)\}\) from the same CFG grammar \(G\), \(S_i\) is the generation context AST slice and \(S'_i\) is the generated result AST slice, \(\psi(D) = \{G_1, G_2, \ldots, G_k\}\) denotes the set of the generators generalized from \(D\).

In Figure 7, \(A\) is the formula for a generator that generates from scratch (i.e., \(b = 0\)) and \(B\) is the formula for a generator that generates from copying the existing AST tree (i.e., \(b = 1\)).

The formula \(A\) produces the generator by computing the bound of the number of nodes and the set of non-terminals in the supplied slices. The formula \(B\) produces the generator by computing 1) the bound of the minimum diff distance between each supplied slice and an arbitrary existing forest in the AST tree and 2) the set of non-terminals of the root node labels of the supplied slices.

**Definition 13** (Transform Generalization). Figure 8 presents the definition of \(\Psi(D)\). Given a set of pairs of AST slices \(D = \{(S_1, S'_1), (S_2, S'_2), \ldots, (S_m, S'_m)\}\) where \(S_i\) is the AST slice before a change and \(S'_i\) is the AST slice after a change, \(\Psi(D)\) denotes the set of transforms generalized from \(D\).

The formula \(\psi\) in Figure 8 invokes \(\Psi\) twice to compute the template AST forest before the change \(T_0\) and the template AST forest after the change \(T_1\). It then computes \(B\) by invoking \(\psi\) to obtain the generalized generators for AST sub-slices that match against each free template variable in \(T_1\).

Note that Figure 9 presents the definition of \(\Psi'\). Intuitively, \(\Psi'\) is the generalization function for template AST forests. The function \(\Psi'(S, M) = \langle T, M' \rangle\) takes a list of AST slices \(S\) and an initial variable binding map \(M\) and produces a generalized template AST forest \(T\) and an updated variable binding map \(M'\).

The first two rows in Figure 9 correspond to the formulas for the cases where each slice is a single tree and the root nodes of the slice trees do not match. The fourth and fifth rows correspond to the formulas for the cases where each slice is a single tree and the root nodes of the slice trees do match. The fourth formula handles the case where


\[ S = \langle S_1, S_2, \ldots, S_m \rangle \]

\[ G = (N, \Sigma, R, s) \]

\[ x' \text{ is a fresh node} \]

\[ v' \text{ is a fresh template variable} \]

\[ \forall i \in \{1, 2, \ldots, m\} : Si = \langle Ti, Li \rangle \]

\[ \text{Li} = \ldots \]

generalized transforms are useful. The goal of the sampling algorithm is to use the described generalization function.

\[ \text{The fifth formula handles the case where there is an existing template variable in } D \text{. The formula creates a template AST forest with a new template variable and updates the variable binding map to include this new variable accordingly.} \]

\[ \text{The sixth row corresponds to the formula for the case where each slice is a forest with the same number of trees. The formula recursively invokes } \Psi \text{ on each individual tree and combines the obtained template AST forests. The seventh row corresponds to the formula for the case in which each slice is a forest, the forests do not match, and there is an existing template variable in } M \text{ to match these forests. The formula therefore creates a template AST forest with the matching variable. The eighth row corresponds to the formula for the case where the forests do not match and there is no template variable in } M \text{ to match these forests. The formula creates a template AST forest with a new template variable and updates the variable binding map accordingly.} \]

\[ \text{Theorem 14 (Soundness of Generalization). For any set of AST slice pairs } D, \forall P \in \Psi(D), \forall(S, S') \in D, (P, S) \Rightarrow \text{slice } S'. \]

The generalization function \( \Psi \) is sound so that the transform \( \mathcal{P} \) is able to generate the corresponding change for each pair in \( D \), from which it is generalized. Intuitively, assume a transform space that denotes all possible program changes and the program changes in the training database \( D \) are points in the transform space. Then the generalization function \( \Psi \) produces a set of potentially useful transforms, each of which covers all of the points for \( D \) in the space.

\[ \text{3.4 Sampling Algorithm} \]

Given a training database \( D \), we could obtain an exponential number of transforms with the generalization function \( \Psi \) described in Section 3.3, i.e., we can invoke \( \Psi \) on any subset of \( D \) to obtain a different set of transforms.

Not all of the generalized transforms are useful. The goal of the sampling algorithm is to use the described generalization function.

<table>
<thead>
<tr>
<th>( \psi(S, M) )</th>
<th>Conditions for ( k ) and ( c )</th>
<th>Other Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (G, \emptyset, \emptyset, \emptyset, \emptyset, M) )</td>
<td>( \forall i \in {1, \ldots, m} ) ( k_i = 0 )</td>
<td></td>
</tr>
<tr>
<td>( (G, V, \gamma, \nu, X', \xi, \sigma, L', \omega, d) )</td>
<td>( \forall i \in {1, \ldots, m} ) ( k_i = 1 ) ( \sigma_d = d )</td>
<td>( S' = \langle (T_1, x_1(1)), (T_2, x_2(2)), \ldots, (T_m, x_m(1)) \rangle )</td>
</tr>
</tbody>
</table>

Figure 9. Definition of \( \psi' \)

Input: a training set of pairs of AST slices \( D \) and a validation set of pairs of AST slices \( E \)

Output: a set of transforms \( \mathcal{P}' \)

1. \( \mathcal{W} \leftarrow \{((S, S'), (S'', S''')) \mid (S, S') \in D, (S'', S''') \in D, (S', S'') \neq (S'', S''')\} \)
2. for \( i = 1 \) to 5 do
3. \( f \leftarrow \text{fitness}(\mathcal{W}, S, D, E) \mid S \in \mathcal{W} \)
4. \( \mathcal{W}' \leftarrow \{S \mid S \in \mathcal{W}, f(S) > 0\} \)
5. Sort elements in \( \mathcal{W}' \) based on \( f \)
6. Select top \( \alpha \) elements in \( \mathcal{W}' \) with largest \( f \) value as a new set \( \mathcal{W}'' \)
7. \( \mathcal{W} \leftarrow \mathcal{W}'' \)
8. if \( i \neq 5 \) then
9. for \( S \in \mathcal{W}'' \) do
10. for \( (S, S') \in D \) do
11. \( \mathcal{W} \leftarrow \mathcal{W} \cup \{S \cup (S, S')\} \)
12. \( \mathcal{P}' \leftarrow U_{S \in \mathcal{W}} \Psi(S) \)
13. return \( \mathcal{P}' \)

Figure 10. Sampling algorithm sample(\( D, E \))
Input: a power set of pairs of AST slices \( W \), a set \( S \in W \), a training set of AST slice pairs \( D \), and a validation set of AST slice pairs \( E \).

Output: the fitness score for \( S \).

for \( S' \) in \( W \) do
  \( A \leftarrow \emptyset \)
  for \( P \) in \( \Psi(S') \) do
    \( B \leftarrow \{(S, S') | (S, S') \in (D \cup E), (P, S) \rightarrow \text{slice } S' \} \)
    \( A \leftarrow A \cup B \)
  end for
  for \( (S, S') \) in \( A \) do
    \( f \leftarrow 0 \)
    for \( P \) in \( \Psi(S) \) do
      \( c \leftarrow \{|\text{str}(T) | (P, S) \rightarrow T|\} \)
      \( d \leftarrow \beta/|C(S, S')| \)
      if \( (S, S') \) in \( D \) then
        \( d \leftarrow d \times \theta \)
      end if
      \( f' \leftarrow f' + \max\{0, d - c\} \)
    end for
    \( f \leftarrow \max\{f, f'\} \)
  end for
return \( f \)

Figure 11. Pseudo-code of fitness\((W, S, D, E)\)

to systematically obtain a set of productive candidate transforms for the inference algorithm to consider.

Figure 10 presents the pseudo-code of our sampling algorithm. As a standard approach in other learning and inference algorithms to avoid overfitting, Genesis splits the training database into a training set \( D \) and a validation set \( E \). Genesis invokes the generalization functions only on pairs in the training set \( D \) to obtain candidate transforms. Genesis uses the validation set \( E \) to evaluate generalized transforms only.

\( W \) in Figure 10 is a work set that contains the candidate subset of \( D \) that the sampling algorithm is considering to use to obtain generalized transforms. The algorithm runs five iterations. At each iteration, the algorithm first computes a fitness score for each candidate subset, keep the top \( \alpha \) candidate subsets (we empirically set \( \alpha = 500 \) in our experiments), and eliminate the rest from \( W \) (see lines 3-7). The algorithm then attempts to update \( W \) by augmenting each subset in \( W \) with one additional pair in \( D \) (see lines 8-10).

Note that it is possible to run more iterations to obtain better candidate patch sets. In practice, we find that the work set \( W \) always converges after five iterations in our experiments. We do not observe any useful transforms that can only be generalized from more than five training AST slice pairs in our experiments.

Figure 11 presents the pseudo-code for the fitness function Genesis uses in its sampling algorithm. The function first computes for each training and validation pair, the number of candidate subsets that produce a transform that covers the pair (see lines 1-8). In this function, a transform covers a AST slice pair if the transform can generate the corresponding change for the slice and the size of the search space derived from this transform is less than a threshold \( \beta \) (see line 5). We empirically set \( \beta = 5 \times 10^4 \) for all experiments we performed.

The algorithm then computes the score for a transform \( P \) as follows. For each validation pair \( (S, S') \) in \( E \) that \( P \) covers, it gets a bonus score \( \max(\beta/|C(S, S')| - c, 0) \), where \( C(S, S') \) is the number of candidate subsets in \( W \) that cover the pair and \( c \) is the size of the number of candidate changes of applying \( P \) to \( S \). The intuition here is to obtain a more diverse set of candidate transforms, i.e., a transform that covers a pair which is also covered many other other transforms should receive much less score than a transform that covers a pair which is not covered by any other transform.

For each training pair \( (S, S') \) in \( D \), the bonus score is \( \max(\beta/|C(S, S')| \times \theta - c, 0) \) instead. We empirically set \( \theta = 0.1 \) in our experiments. The intuition here is that the pairs in the training data should provide much less scores than the pairs in validation set to avoid overfitting.

There are three optimizations in the Genesis implementation for the above sampling algorithm. Firstly, Genesis filters many unproductive subsets that may yield intractable search space when introducing new training in \( W \) at lines 10-11. For a new subset \( A \), if another subset \( B \subseteq A \) was already discard because the space sizes of obtained transforms from \( B \) are more than \( \beta \), then Genesis discards \( A \) immediately without adding it to \( W \) at line 11.

Secondly, the computation of the search space size of the transform at line 5 and line 14 in Figure 11. Genesis computes an estimated size instead of generating each AST tree one by one. This estimation assumes that all generators in the transform generate binary trees. This optimization trades the accuracy of the fitness score computation for performance. Thirdly, Genesis computes \( C \) in Figure 11 at the start of each iteration of the sampling algorithm to avoid redundant computation during each invocation of the fitness() function.

3.5 Search Space Inference Algorithm

ILP Formulation: Given a candidate set of transforms \( \mathcal{P}' \), the goal is to select a subset \( \mathcal{P} \) from \( \mathcal{P}' \) to form the result search space. We formulate the trade-off of the search space design between the coverage and the tractability as an integer linear programming (ILP) problem.

Figure 12 presents the ILP formulation of the transform selection problem. \( C_{ij} \) corresponds to the space size derived from the \( j \)-th transform when applying to the \( i \)-th AST slice pair in the validation set. \( G_{ij} \) indicates whether the space derived from the \( j \)-th transform contains the corresponding change for the \( i \)-th AST slice pair.
Input: a set of training AST slice pairs \( D \)
Output: a set of transforms that generate a search space

1. Remove a subset of \( D \) from \( D \) to form the validation set \( E \)
2. \( P' \) ← sample(\( D, E \))
3. Solve ILP in Figure 12 to obtain \( P \)
4. return \( P \)

**Figure 13.** Search space inference algorithm

The variable \( x_i \) indicates whether the result search space covers the \( i \)-th AST slice pair and the variable \( y_j \) indicates whether the ILP solution selects the \( i \)-th transform. The ILP optimization goal is to maximize the sum of \( x \), i.e., the total number of covered AST pairs in the validation set.

The first group of constraints is for tractability. The \( i \)-th constraint specifies that if the derived final search space size (i.e., \( \sum_{j=1}^{k} c_{ij} y_j \)), when applied to the \( i \)-th AST pair in \( E \), should be less than \( \beta \) if the space covers the \( i \)-th AST pair (i.e., \( x_i = 1 \)) or less than \( \zeta \) if the space does not cover the \( i \)-th AST pair (i.e., \( x_i = 0 \)). In Genesis, \( \beta = 5 \times 10^3 \) and \( \zeta = \infty \).

The second group of constraints is for coverage. The \( i \)-th constraint specifies that if the final search space covers the \( i \)-th AST slice pair in \( E \) (i.e., \( x_i = 1 \)), then at least one of the selected transforms should cover the \( i \)-th pair.

Genesis also implements an alternative ILP formulation which considers also the patches in the training set and maximizes the number of covered training patches when two different solutions have the same number of covered validation patches. We empirically find that this alternative formulation tends to produce better search spaces when the validation set is small.

**Inference Algorithm:** Figure 13 presents the high-level pseudo-code of the Genesis inference algorithm. Starting from a training set of AST slice pairs \( D \), Genesis first removes 25% of the AST slice pairs from \( D \) to form the validation set \( E \). It then runs the sampling algorithm to produce a set of candidate transforms \( P' \). It finally solves the above ILP with Gurobi [16], an off-the-shelf solver, to obtain the set of transforms \( P \) that forms the result search space.

### 3.6 Java Implementation

We have implemented the Genesis inference algorithm for Java programs. We use the spoon library [32] to parse Java programs to obtain Java ASTs. We next discuss several extensions of the above inference algorithm for handling Java programs.

**Semantic Checking:** Genesis performs type checking in its implementation of the generation operators for generators and transforms. Genesis will discard any AST tree or AST forest that cannot pass Java type checking. Genesis also performs semantic checking to detect common semantic problems like undefined variables, uninitialized variables, etc.

**Identifiers and Constants:** The CFG for Java has an infinite set of terminals, because there are an infinite amount of possible variables, fields, functions, and constants. For the kind of generators that enumerate all possible AST forests (i.e., \( b = 0 \)), Genesis does not generate changes that import new packages and or changes that introduce new local variables (even if a change introduces new local variables, it is typically possible to find a semantic equivalent change that does not). Therefore Genesis only considers a finite set of possible variables, fields, and functions.

Genesis also extends enumeration-based generators so that each generator has an additional set to track the constant values that the generator can generate. For the generation operator of a generator,

Input: the original program \( p \), the validation test suite \( V \), and the set of the transformation patterns of the search space \( P \)
Output: the list of generated patches \( S \)

1. \( S \) ← localization(\( p, V \))
2. \( G \) ← emptylist
3. for \( S \) in \( S \) do
4. for \( P' \) in \( P \) do
5. for \( p' \) in \{str\( (T) \mid \{P, S\} \implies T \}\) do
6. if validate\( (p', V) \) then
7. Append \( p' \) to \( G \)
8. return \( G \)

**Figure 14.** Patch generation algorithm

Genesis will only consider finite constant values that are 0, 1, \text{null}, \text{false}, or any value that is inside the tracked set of the generator.

Many string constants in Java programs are text messages (e.g., the message in throw statements). These constants may cause a sparsity problem when Genesis computes the set of the allowed constants during the generalization process. To avoid such sparsity problems, Genesis detects such string constants and convert them to a special constant string — the specific string values are typically not relevant to the overall correctness of the programs.

**Identifier Scope:** Genesis exploits the structure of Java programs to obtain more accurate generators. Each enumeration-based generator (i.e., \( b = 0 \)) tracks separate bounds for the number of variables and functions it uses inside the original slice, from the enclosing function, from the enclosing file, and from all imported files. For example, a generator may specify that it will only use up to two variables from the enclosing function in the generated AST forests. Similarly, each copy-based generator (i.e., \( b = 0 \)) has additional flags to determine whether it copies code from the original code, the enclosing function, or the entire enclosing source file.

**Code Style Normalization:** Genesis has a code style normalization component to rewrite programs in the training set while preserving semantical equivalence. The code style normalization enables Genesis to find more common structures among ASTs of training patches and improves the quality of the inferred transforms and search spaces.

### 4. Patch Generation

Figure 14 presents the Genesis patch generation algorithm. The Genesis error localization algorithm (line 1) produces a ranked list of suspicious locations (as AST snippets) in the original program \( p \). Genesis applies each transform in \( P \) to each suspicious AST snippet \( S \) in \( S \) to obtain candidate patches \( p' \) (lines 3-5). It validates each candidate patch against the test cases and appends it to the returned patch list if it passes all test cases (lines 6-7). Our current implementation supports any Java application that operates with the Apache maven project management system [1] and JUnit [5] testing framework.

Genesis is designed to work with arbitrary error localization algorithms. Our current implementation starts with stack traces generated from test cases that trigger the null pointer or out of bounds access error. It extracts the top ten stack trace entries and discards any entries that are not from source code files in the project (as opposed to external libraries or JUnit). For each entry it finds the corresponding line of code in a project source code file and collects that line as well as the 50 lines before and after that line of code.
For each of the collected lines of code, Genesis first computes a suspiciousness score between 0 and 0.5. The line of code given by the first stack trace entry has a suspiciousness score of 0.5, with the score linearly decreasing to zero as the sum of the distance to the closest line of code from the stack trace and the rank of that line within the stack trace increases. Genesis prioritizes lines containing if, try, while, or for statements by adding 0.5 to their suspiciousness scores. The final scores are in the range of 0 to 1. Genesis prioritizes lines of code for patch generation according their final scores.

5. Experimental Results

We next present experimental results of Genesis.

5.1 Methodology

Collect NPE and OOB Patches and Errors: We developed a script that crawled the top 1000 github Java projects (ranked by number of stars) to collect 503 null pointer error patches from 126 different applications. The script also crawled the top 1000 github Java projects and a list of 50968 github repositories from the MUSE corpus [7] to collect 212 out of bounds error patches from 117 different applications. The script that crawls the repositories and collects a project revision if 1) the project uses the apache maven management system [1], 2) we can use maven 3.3 to automatically compile both the current revision and the parent revision of the current revision (in the github revision tree) in our experimental environment, 3) we can use the spoon library [32] to parse the source code of both of the two revisions into AST trees, 4) the commit message of the current revision contains certain keywords to indicate that the revision corresponds to a patch for NPE errors or OOB errors, and 5) the revision changes only one source file (because revisions that change more than one source file often correspond to composite changes and not just patches for NPE or OOB errors).

For NPE errors, the scripts search for keywords “null deref”, “null pointer”, “null exception”, and “npe”. For OOB errors, the scripts search for keywords “out of bounds”, “bound check”, “bound fix”, and “oob”. We manually inspected the retrieved revisions to discard revisions that do not correspond to actual NPE or OOB errors. Note that we discard many repositories and revisions because we are unable to automatically compile them with maven, i.e., they do not support maven 3.3 or they have special dependencies that cannot be automatically resolved by the maven system.

Benchmark Errors: We then went over each of the collected patches with another script we developed. The script collects a revision if 1) the revision has a JUnit [5] test suite in the repository that Genesis can run automatically, 2) the JUnit test suite contains at least one test case that can expose and reproduce the error in our experimental environment, 3) the JUnit test suite contains at least 50 test cases in total, and 4) the test suite does not cause non-deterministic behaviors. This script collects 20 NPE errors and 13 OOB errors from the total 503 NPE and 212 OOB errors, respectively. These are the benchmark errors.

Partition into Training, Validation, and Benchmark Patches: We partition the collected 503 NPE and 212 OOB patches into training, validation, and benchmark patches as follows. We first removed the 20 NPE and 13 OOB benchmark errors, leaving 483 and 199 remaining patches. We partitioned these remaining patches into 362 NPE and 149 OOB training patches and 121 NPE and 50 OOB validation patches (the inference algorithm uses these patches to avoid overfitting, see Section 3.5).

Search Space Inference: We run the Genesis search space inference algorithm on the NPE training and validation patches to infer the NPE transforms and on the OOB training and validation patches to infer the OOB transforms. For the OOB errors, we use the alternative ILP formulation in Section 3.5 because the number of validation OOB patches is small. We run the inference algorithm with 36 threads in parallel on an Amazon EC2 c4.8xlarge instance with Intel Xeon E5-2666 processors, 36 vCPU, and 60GB memory.

Table 1. Transform and search space inference results

<table>
<thead>
<tr>
<th></th>
<th>NPE</th>
<th>OOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transforms after Sampling</td>
<td>985</td>
<td>461</td>
</tr>
<tr>
<td>Covered Training Patches after Sampling</td>
<td>220 of 362</td>
<td>85 of 149</td>
</tr>
<tr>
<td>Covered Validation Patches after Sampling</td>
<td>58 of 121</td>
<td>14 of 50</td>
</tr>
<tr>
<td>Final Inferred Transforms</td>
<td>14</td>
<td>23</td>
</tr>
<tr>
<td>Covered Training Patches in Search Space</td>
<td>130 of 362</td>
<td>71 of 149</td>
</tr>
<tr>
<td>Covered Validation Patches in Search Space</td>
<td>51 of 121</td>
<td>10 of 50</td>
</tr>
<tr>
<td>Total Inference Time</td>
<td>199m</td>
<td>65/m</td>
</tr>
</tbody>
</table>

5.2 Inference Results

Table 1 presents search space inference results. The Genesis sampling algorithm (Section 3.4) produces 985 NPE transforms and 461 OOB transforms. These transforms cover 220 of the NPE training patches, 58 of the NPE validation patches, 85 of the OOB training patches, and 14 of the OOB validation patches. The Genesis search space inference (Section 3.5) algorithm selects 14 of the 985 NPE transforms and 23 of the 461 OOB transforms. These selected transforms cover 130 of the NPE training patches, 51 of the NPE validation patches, 71 of the OOB training patches, and 10 of the OOB validation patches. The search space inference times are reasonable at 199 minutes for NPE inference and 657 minutes for OOB inference. The time is dominated by the sampling algorithm. Solving the ILPs takes less one minute for both NPE and OOB transforms. We attribute the larger OOB inference time to the fact that the inferred OOB transforms have more sophisticated generators that tend to generate larger, more complex search spaces.

5.3 Patch Generation Results

Table 2 presents the Genesis patch generation results for the benchmark 20 NPE and 13 OOB errors. There is a line in the table for each benchmark error. The Init. Time column presents the amount of time required to initialize the search for that error. The Search
Space Size column presents the size of the Genesis search space for that error, the Explored Space Size column presents the size of the search space that the algorithm explores within the five hour timeout (for most errors Genesis is able to explore the full search space). Validated Patches presents the number of candidate patches that validate (produce correct outputs for all test cases), and Correct Patches presents the number of validated patches that are correct. We note that some errors have multiple correct patches. These are different patches that are semantically equivalent in the context in which they appear.

The last three columns present statistics for the first generated correct patch, specifically how long it takes to generate the patch (Generation Time), the rank of the first correct patch in the sequence of validate patches, and the rank of the correct patch in the sequence of candidate patches. The generated NPE patches used a total of 10 of the 14 inferred NPE transforms; the generated OOB patches used a total of 11 of the 23 inferred OOB transforms.

To isolate the effect of error localization, we also run Genesis with an oracle that identifies, for each error, the correct line of code to patch. For NPE errors the oracle results are close to unchanged (for HikariCP the Validated Rank changes from 12 to 1; for javaslang the Validate Rank changes from 3 to 5), which highlights the effectiveness of our error localization algorithm for this class of errors. The Validate Rank for javaslang is slightly worse with oracle error localization because of traversal order variations in the underlying set and map data structures in the patch generation algorithm.

For OOB errors oracle error localization enables Genesis to generate 2 more correct patches, improves the Validation Rank of two patches (httpcore and vectorz) from 60 and 10 to 1 and changes the Validation Rank of wicket from 7 to 8 (again because of traversal order variations in the underlying set and map data structures). The remaining unpatched errors all fall outside the inferred search spaces. We believe that an enhanced training set would enable Genesis to learn transforms that bring most if not all of these errors within the Genesis search space.

**NPE Patch Categories:** All of the generated correct patches are available in the supplemental material. Broadly speaking, the generated correct NPE patches fall within several categories. Some either 1) return a synthesized value or void, 2) throw a new synthesized exception, or 3) skip a statement with a null pointer error. Other patches modify an existing boolean expression, either by conjoning or disjoning a synthesized expression or by replacing a clause in the expression with a pointer equality check.

**Table 2. Experimental results for automatic patch generation.**

<table>
<thead>
<tr>
<th>Repository</th>
<th>Revision</th>
<th>Type</th>
<th>Init. Time</th>
<th>Search Space Size</th>
<th>Explored Space Size</th>
<th>Search Time</th>
<th>Validated Patches</th>
<th>Correct Patches</th>
<th>First Correct Patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>caelum-stella</td>
<td>2e5c5459</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>2682</td>
<td>2682</td>
<td>6m</td>
<td>4</td>
<td>1</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>caelum-stella</td>
<td>2d349c6</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>1287</td>
<td>1287</td>
<td>8m</td>
<td>18</td>
<td>18</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>caelum-stella</td>
<td>e73113f</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>1320</td>
<td>1320</td>
<td>8m</td>
<td>18</td>
<td>18</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>HikariCP</td>
<td>ce4f92</td>
<td>NPE</td>
<td>3m</td>
<td>7262</td>
<td>7262</td>
<td>74m</td>
<td>26</td>
<td>2</td>
<td>47m</td>
</tr>
<tr>
<td>nutz</td>
<td>80e83d0</td>
<td>NPE</td>
<td>1m</td>
<td>21266</td>
<td>21266</td>
<td>120m</td>
<td>14</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>spring-data-rest</td>
<td>aa28ae6</td>
<td>NPE</td>
<td>5m</td>
<td>5067</td>
<td>5576</td>
<td>&gt;5h</td>
<td>8</td>
<td>6</td>
<td>38m</td>
</tr>
<tr>
<td>checkstyle</td>
<td>8381754</td>
<td>NPE</td>
<td>2m</td>
<td>23261</td>
<td>23261</td>
<td>227m</td>
<td>4</td>
<td>4</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>checkstyle</td>
<td>5368c20</td>
<td>NPE</td>
<td>2m</td>
<td>25984</td>
<td>25984</td>
<td>237m</td>
<td>8</td>
<td>8</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>checkstyle</td>
<td>aa4606c</td>
<td>NPE</td>
<td>2m</td>
<td>25752</td>
<td>25752</td>
<td>225m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>checkstyle</td>
<td>aa8294d</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>0</td>
<td>0</td>
<td>&lt;1m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>jongo</td>
<td>f46658</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>10893</td>
<td>10893</td>
<td>175m</td>
<td>3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Dataflow</td>
<td>co6125d</td>
<td>NPE</td>
<td>3m</td>
<td>4039</td>
<td>4039</td>
<td>43m</td>
<td>1</td>
<td>1</td>
<td>2m</td>
</tr>
<tr>
<td>webmagic</td>
<td>f21588</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>8398</td>
<td>8398</td>
<td>45m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>javapet</td>
<td>70b38ce5</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>8107</td>
<td>8107</td>
<td>47m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>closure-compiler</td>
<td>9828574</td>
<td>NPE</td>
<td>3m</td>
<td>116853</td>
<td>58520</td>
<td>&gt;5h</td>
<td>4</td>
<td>4</td>
<td>6m</td>
</tr>
<tr>
<td>truth</td>
<td>996314c</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>1328</td>
<td>1328</td>
<td>3m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>error-prone</td>
<td>3709338</td>
<td>NPE</td>
<td>2m</td>
<td>40238</td>
<td>40238</td>
<td>294m</td>
<td>5</td>
<td>1</td>
<td>73m</td>
</tr>
<tr>
<td>javaslang</td>
<td>fa9ac2</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>54224</td>
<td>54224</td>
<td>63m</td>
<td>5</td>
<td>1</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>Activiti</td>
<td>3d624a5</td>
<td>NPE</td>
<td>2m</td>
<td>20057</td>
<td>1632</td>
<td>&gt;5h</td>
<td>108</td>
<td>2</td>
<td>7m</td>
</tr>
<tr>
<td>spring-hateoas</td>
<td>7487497</td>
<td>NPE</td>
<td>&lt;1m</td>
<td>1064</td>
<td>1064</td>
<td>3m</td>
<td>6</td>
<td>6</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>Bukkit</td>
<td>a91c46c</td>
<td>OOB</td>
<td>&lt;1m</td>
<td>657814</td>
<td>657814</td>
<td>181m</td>
<td>7</td>
<td>5</td>
<td>&lt;1m</td>
</tr>
<tr>
<td>RoaringBitmap</td>
<td>29c0d59</td>
<td>OOB</td>
<td>4m</td>
<td>407858</td>
<td>45604</td>
<td>&gt;5h</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>commons-lang</td>
<td>52b967</td>
<td>OOB</td>
<td>1m</td>
<td>49561</td>
<td>3038</td>
<td>&gt;5h</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>HdrHistogram</td>
<td>db18013</td>
<td>OOB</td>
<td>&lt;1m</td>
<td>97613</td>
<td>97613</td>
<td>243m</td>
<td>71</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>spring-hateoas</td>
<td>29b4334</td>
<td>OOB</td>
<td>&lt;1m</td>
<td>16506</td>
<td>16506</td>
<td>17m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>wicket</td>
<td>b7082e2b</td>
<td>OOB</td>
<td>5m</td>
<td>93738</td>
<td>93738</td>
<td>274m</td>
<td>22</td>
<td>6</td>
<td>79m</td>
</tr>
<tr>
<td>coveralls-maven-plugin</td>
<td>20490f0</td>
<td>OOB</td>
<td>&lt;1m</td>
<td>1567</td>
<td>1567</td>
<td>3m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>named-regexp</td>
<td>828de6b</td>
<td>OOB</td>
<td>&lt;1m</td>
<td>0</td>
<td>0</td>
<td>&lt;1m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>jgit</td>
<td>929862f</td>
<td>OOB</td>
<td>2m</td>
<td>60921</td>
<td>60921</td>
<td>121m</td>
<td>6</td>
<td>6</td>
<td>55m</td>
</tr>
<tr>
<td>jPOS</td>
<td>df400ac</td>
<td>OOB</td>
<td>2m</td>
<td>88610</td>
<td>88610</td>
<td>176m</td>
<td>9</td>
<td>2</td>
<td>8m</td>
</tr>
<tr>
<td>httpcore</td>
<td>dd08a9e</td>
<td>OOB</td>
<td>2m</td>
<td>123402</td>
<td>70176</td>
<td>&gt;5h</td>
<td>104</td>
<td>9</td>
<td>148m</td>
</tr>
<tr>
<td>vectorz</td>
<td>229180d</td>
<td>OOB</td>
<td>&lt;1m</td>
<td>108273</td>
<td>108273</td>
<td>181m</td>
<td>25</td>
<td>9</td>
<td>60m</td>
</tr>
<tr>
<td>maven-shared</td>
<td>77937c1</td>
<td>OOB</td>
<td>2m</td>
<td>0</td>
<td>0</td>
<td>&lt;1m</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>
OOB Patch Categories: The generated correct OOB patches also fall within several categories. Some patches introduce a comparison that checks for an out of bounds access followed by a return or break statement if the comparison succeeds. Other patches modify an existing boolean expression, typically by conjoining or disjoining a synthesized expression or by changing a binary comparison operator (such as changing < to <=).

6. Related Work
Generate And Validate Systems: Generate and validate patch generation systems apply a set of transforms to generate a search space of candidate patches that are then evaluated against a set of inputs to filter out patches that produce incorrect outputs for the test inputs. Prophet [22] and SPR [23] apply a set of predefined parameterized transformation schemas to generate candidate patches. Prophet processes a corpus of successful human patches to learn a model of correct code to rank plausible patches; SPR uses a set of hand-code heuristics for this purpose. GenProg [21, 46], AE [45], and RSRepair [35] use a variety of search algorithms (genetic programming, stochastic search, random search) in combination with transforms that delete, insert, and swap existing program statements. Kali [36] applies a single transform that simply deletes code. All of these systems were evaluated on the same benchmark set [21]. For the 69 defects in this set (the set also contains 36 functionality changes), Prophet, SPR, Kali, GenProg, RSRepair, and AE generate correct patches for 15, 11, 2, 1, 2, and 2 defects, respectively. We attribute the relatively poor performance of Kali, GenProg, RSRepair, and AE to the fact that their search spaces do not appear to contain correct patches for the remaining defects in the set [23, 36]. Like Prophet, history-driven program repair [20] uses information from previous human patches to rank candidate patches generated by human-specified transforms.

PAR [18] deploys a set of patterns to fix bugs in Java programs, with the patterns manually derived by humans examining multiple real-world patches. The PAR null pointer checker pattern inserts if statements that either 1) skip a statement with a null dereference or 2) returns a default value before a null dereference. The PAR range checker pattern inserts bounds checks. Genesis automatically infers a larger and richer set of transforms that generate all of the patches generated by the PAR manually-derived patterns and more. In particular, the generated Genesis patch in Section 2 is outside the PAR search space.

Genesis differs from all of these systems in that it does not work with a fixed set of human-specified transforms. It instead automatically processes patches from repositories to automatically infer a set of transforms that together define its patch search space.
Constraint Solving Systems: Prophet [22], SPR [23], Qlose [11], NOPOL [15], SemFix [31], and Angelix [27] all use constraint solving to generate new values for potentially faulty expressions (often faulty conditions). ClearView [34] enforces learned invariants to eliminate security vulnerabilities. Angelic Debugging [10] finds new values for potentially incorrect subexpressions that allow the program to produce correct outputs for test inputs.

PHPQuickFix and PHPRepair use string constraint-solving techniques to automatically repair PHP programs that generate HTML [42]. By formulating the problem as a string constraint problem, PHPRepair obtains sound, complete, and minimal repairs to ensure the patched php program passes a validation test suite. Specification-based data structure repair [12, 13, 17, 47] takes a data structure consistency specification and an inconsistent data structure, then synthesizes a repair that produces a modified data structure that satisfies the consistency specification.

Genesis differs from all of these systems in that it works with automatically inferred transforms, with generators playing the role of constraint solvers to generate expressions that enable parameterized transforms to produce correct patches.
Repair with Formal Specifications: It is possible to leverage formal specifications to generate patches that produce a patched program that satisfies the specification [19, 33, 41]. One difference is that Genesis works with large real world applications where formal specifications are typically not available.

Probabilistic Model for Programs: There is a rich set of work on applying probabilistic model and machine learning learning techniques for programs, specifically, for identifying correct repairs [22], code refactoring [38], and code completion [9, 37, 39]. These techniques learn a probabilistic model from a training set of patches or programs and then use the learned model to identify the best repair or token for a defective or partial program. In contrast, instead of learning individual concrete patches, Genesis has the high-order goal of inferring transforms that can be applied to a new bug to generate a set of candidate patches. Genesis does not use probabilistic models. It instead obtains candidate transforms with a novel generalization algorithm and formulates the transform selection problem as an integer linear programming.

Repair Model Mining: Martinez and Monperrus manually analyze previous human patches to mine repair models for program repair systems. They manually define a set of transforms and then classify the patches into the defined transforms based on the kind of modification operations of the patches [26]. In contrast, Genesis does not work with any predefined transform and automatically infers the set of transforms from a set of human patches.

Systematic Edit: SYDIT [29] and Lase [30] extract edit scripts from one (SYDIT) or more (Lase) example edits. The script is a sequential list of modification operations that insert statements or update existing statements. SYDIT and Lase then generate changes to other code snippets in the same application with the goal of automating repetitive edits. RASE [28] uses Lase edit scripts to refactor code clones. FixMeUp [44] works with access control templates that implement policies for sensitive operations. Using these templates, FixMeUp finds unprotected sensitive operations and inserts appropriate checks. An analysis of the application can extract an application-specific template [43], which FixMeUp can then apply across the same application. Genesis differs in that it processes multiple patches from multiple applications to derive generalized application-independent transforms that it can apply to fix bugs in yet other applications. The Genesis transforms include generators so that transforms can generate multiple candidate patches (as opposed to a single edit as in SYDIT, Lase, and FixMeUp).

Dynamic Recovery: Failure-oblivious computing [40] discards out of bounds writes and manufactures values for out of bounds reads. RCV [25] returns zero as the result of null pointer dereferences and divide by zero errors. APPEND [14] detects attempted null pointer dereferences and applies recovery actions such as creating a default object to replace the null pointer. In all cases the goal is to enable successful (but not necessarily correct) continued execution.

Genesis, in contrast, learns transforms and applies these transforms to derive a patched program without the null pointer or out of bounds access error — the goal is to obtain a correct patch, not simply continued execution via run-time recovery. The automatically inferred Genesis templates provide a broader and more sophisticated set of techniques for dealing with null pointer dereferences and out of bounds accesses.
7. Conclusion

Previous generate and validate patch generation systems work with a fixed set of transforms defined by their human developers. By automatically inferring transforms from sets of successful human patches, Genesis makes it possible to leverage the combined expertise and patch generation strategies of developers worldwide to automatically patch bugs in new applications. Results from our implemented Genesis system highlight its ability to infer productive search spaces and deploy those search spaces to effectively patch null pointer and out of bounds errors in real-world Java programs.

References


