StaticFixer: From Static Analysis to Static Repair

ANONYMOUS AUTHOR(S)

Static analysis tools are traditionally used to detect and flag programs that violate properties. We show that static analysis tools can also be used to perturb programs that satisfy a property to construct variants that violate the property. Using this insight we can construct paired data sets of unsafe-safe program pairs, and learn strategies to automatically repair property violations. We present a system called StaticFixer, which automatically repairs information flow vulnerabilities using this approach. Since information flow properties are non-local (both to check and repair), StaticFixer also introduces a novel domain specific language (DSL) and strategy learning algorithms for synthesizing non-local repairs. We use StaticFixer to synthesize strategies for repairing two types of information flow vulnerabilities, unvalidated dynamic calls and cross-site scripting, and show that StaticFixer successfully repairs several hundred vulnerabilities from open source JavaScript repositories, outperforming neural baselines built using CodeT5 and Codex.

1 INTRODUCTION

Static analysis (SA for brevity) takes a program P and a property φ as inputs, and checks if the program satisfies the property. If the program violates the property, SA outputs an error report, which the developer uses to fix the violation. In this paper, we present a system called StaticFixer which automates the process of fixing these violations.

Given a SA tool, a property φ, and a large corpus of programs $\mathcal{P}$ with a sufficient variety of programs that satisfy φ, our system StaticFixer automatically learns to fix static analysis violations in a data-driven manner. StaticFixer consists of two stages:

(1) Data collection: This stage uses the corpus $\mathcal{P}$ to construct a set of "paired" programs $\{(P_1, P'_1), (P_2, P'_2), \ldots, (P_n, P'_n)\}$, such that, for each pair (P_i, P'_i), the first program P_i violates the property φ, the second program P'_i contains the fix for the violation. Furthermore, except for the fix for the violation of φ, we have that P_i and P'_i are identical.

(2) Strategy learning: This stage uses the paired set of programs above as a training set to automatically learn a repair strategy S for fixing the violation. Specifically, for any program, P that violates φ and roughly resembles one or more programs in $\mathcal{P}$, the goal is for $S(P)$ to fix the violation of φ.

We consider the class of information flow safety properties. Violating these properties can result in the system becoming vulnerable to information flow attacks, allowing malicious user input to flow from an untrusted *source* (e.g., a server request, socket message, file upload) to a sensitive *sink* (e.g., dynamic function execution, shell command, database query). SQL Injection [9], cross-site scripting [3], and prototype pollution [12] are all examples of information flow vulnerabilities. A common strategy to defend against such violations is to use *sanitizers* and *guards* in the code to block such bad flows. Sanitizers and guards ensure that only safe information reaches the sensitive sinks. Information flow safety is a non-local property and both checking and fixing violations require non-local analysis.

We introduce **static-analysis witnessing**, a new technique to collect a high-quality paired dataset of unsafe and safe programs. Instead of using the SA tool to flag programs with violations, we use the internal information captured by the SA tool on programs that satisfy the property, and identify the *reason* why the property is satisfied as a *witness*. In the case of information flow safety properties, these witnesses are usually sanitizers and guards in programs, which break the flow between untrusted sources and a trusted sinks. By identifying and removing the witness, we introduce a violation and convert the safe program to an unsafe program, enabling us to construct safe-unsafe program pairs from programs that satisfy the property.

```
var actions = new Map();
loadActions(actions);
app.get('/run', (req, res) => {
  var action = actions.get(req.action);
  if (action && typeof action === 'function'){
    action(req.inp);
  }
}
```

(a) Safe program I for UDC-TypeCheck vulnerability

```
var actions = new Map();
loadActions(actions);
app.get('/run', (req, res) => {
  var action = actions.get(req.action);
  if (typeof action !== 'function')
    return;
  action(req.inp);
}
```

(b) Safe program II for UDC-TypeCheck vulnerability

```
var actions = new Map();
loadActions(actions);
app.get('/run', (req, res) => {
  var action = actions.get(req.action);
  (typeof action === 'function') && action(req.inp);
}
```

(c) Safe program III for UDC-TypeCheck vulnerability

```
var actions = new Map();
loadActions(actions);
app.get('/run', (req, res) => {
  var action = actions.get(req.action);
  action(req.inp);
}
```

(d) Unsafe program for UDC-TypeCheck vulnerability

Fig. 1. *Witnessing-safe programs* that will be detected by SA-witnessing in (a), (b), and (c). The witnesses making programs safe are highlighted in yellow. (d) depicts an unsafe program will be detected by typical SA.

Consider the code snippets shown in Figure 1. In these code snippets, the value of `req` comes from an untrusted source, and executing `action` (which is dynamically derived from `actions` map using `req.action`) can result in attacks. Programs (a), (b), and (c) are all examples of safe programs that satisfy the property, and the witnesses are highlighted. Program (d) is an unsafe program. By removing the witnesses, we transform each of the safe programs into unsafe variants and use these variants to construct safe-unsafe program pairs for the data collection phase.

For the strategy learning step, we use the paired set of programs as a training set and synthesize repair-strategies in a domain-specific-language (DSL). We build on prior work in synthesizing program transformations from paired programs [15, 34, 44]. However, information flow vulnerabilities are non-local, and repairing them is beyond the scope of previous work. Therefore, we design both a novel DSL and a strategy learning algorithm that is able to learn strategies for such non-local repairs. Consider the example shown in Figure 2. The program in Figure 2a is vulnerable since there is a flow from the untrusted `JSON.Parse` statement in line 12 to the trusted method invocation in line 6 without an intervening sanitizer. The repair here involves the introduction of the guard `if (handlers.hasOwnProperty(data.id))` in line 5, as shown in Figure 2b. Learning such a repair involves (1) learning the location to introduce the repair (which is line 5 in this case), (2) learning the template of the guard that needs to be introduced (which is of the form `REF1.hasOwnProperty(REF2)`, where `REF1` and `REF2` are references, and (3) learning that `REF1` and `REF2` need to be materialized to `handlers` and `data.id` based on the given program. Our DSL and strategy-learning algorithms (described in Section 5) are novel, and are able to learn such intricate and non-local repairs, which involve analyzing both control and dataflows in the program.

Prior works in static repair [14, 15, 25, 28, 29, 34, 35, 45] make simplifying assumptions. First, they assume the availability of paired unsafe-safe program versions from version control. Second, the scope of repairs they consider are typically local. Third, they consider statically typed languages like Java. We focus on information flow vulnerabilities, which are inherently non-local, and we consider JavaScript, which lacks static types. Moreover, we don't make the assumption that we have access to pairs of unsafe and safe programs. Instead, a single static snapshot of source-code repositories is sufficient for our technique. Other approaches [30, 41] in automatic program repair (APR) use program execution on vulnerability-causing examples which do not apply to repairing

Term	Definition
source	a variable whose value is directly set by an (untrusted) user
tainted variable	a variable whose value is derived from a source variable and is controllable by an (untrusted) user; the value of such a variable is called tainted value
sink	a program execution performing a security-critical operation using an input
sanitizer	function that takes the tainted variable as input and removes the taint
guard	a check performed on the tainted values to block execution on malicious inputs

Table 1. information flow vulnerability terminologies and definitions

static vulnerabilities. Crafting repair templates manually [17, 19, 27, 36] is also challenging given the semantic nature of repairs.

We implemented the above approach consisting of static-analysis witnessing and strategy learning steps in a system StaticFixer. In Section 6, we present an empirical evaluation of StaticFixer with code from several hundred JavaScript open source repositories on Github. We use CodeQL [13] as our SA tool, and consider two specific instances of information flow vulnerabilities: (1) unvalidated dynamic call (UDC), and (2) cross-site scripting (XSS). For both these instances, we use static analysis witnessing and witness removal to generate unsafe-safe pairs, and learn repair strategies from this data. Then we evaluate the effectiveness of the learned repair strategies on all the violations detected by CodeQL in these repositories. Thus, our training set is generated from correct coding patterns (which is that static analysis witnessing uses), and our validation set is the set of violations in these repositories (which are disjoint from the training set). We find that StaticFixer is able to correctly repair: (1) **310** vulnerabilities with a success rate of **93.94%**, in the case of unvalidated dynamic call, and (2) **617** vulnerabilities with a success rate of **91.82%**, in the case of cross-site scripting. We compare the results with two neural baselines, namely one obtained by finetuning CodeT5 [38] and the other obtained by few-shot prompting Codex[16] with the same training set as StaticFixer. We find that StaticFixer outperforms both these neural baselines.

To summarize, our main contributions are:

- A new approach called static-analysis witnessing to produce unsafe-safe code pairs from programs that satisfy a property, and
- A novel DSL and strategy learning algorithm to learn non-local repair strategies from paired data sets (such as the ones generated from static analysis witnessing, or other approaches)

We present an implementation of the approaches in a tool, StaticFixer. Our empirical results show that StaticFixer is able to correctly repair hundreds of violations in open source JavaScript repositories, while outperforming neural baselines. We will publicly release the datasets used in our evaluation (Section 6).

2 BACKGROUND AND OVERVIEW

2.1 Problem Background and Motivating Examples

In the previous section we defined information flow vulnerabilities as flow of information from an untrusted source to a trusted or sensitive sink without appropriate sanitization. In Table 1 we define these terms more precisely. SA tools find these vulnerabilities by detecting sources and sinks, and then performing a dataflow analysis between them. We call a program unsafe or safe depending on whether SA detects a violation or not. Further, we call a program a *witnessing-safe program* if it is safe because a *witness*, i.e., a sanitizer or a guard, blocks the flow between a source and a sink.

```
1  handlers["run"] = function (data){
2      ...
3      let foo = handlers[data.id];
4      ...
5
6      foo(data);
7
8  }
9  ...
10 var commHandler = function (event){
11     ...
12     var data = JSON.parse(event.data);
13     handlers["run"](data);
14 }
```

(a) Example for vulnerable program

```
1  handlers["run"] = function (data){
2      ...
3      let foo = handlers[data.id];
4      ...
5  +   if handlers.hasOwnProperty(data.id){
6          foo(data);
7  +   }
8  }
9  ...
10 var commHandler = function (event){
11     ...
12     var data = JSON.parse(event.data);
13     handlers["run"](data);
14 }
```

(b) Example for fixed safe program

Fig. 2. Example and corresponding fix for the UDC-MembershipCheck vulnerability.

Note that a safe program is not necessarily a witnessing-safe program. In particular, a program that does not have any sources or sinks is vacuously safe, but not a witnessing-safe program. Next, we give two simplified examples with violations and show how one can fix them.

Example 1. Figure 2a is an unsafe program containing the UDC-MembershipCheck vulnerability where an untrusted user input is used as a key to index into a record of functions. A malicious user can exploit this vulnerability by passing missing keys or keys defined by parent or base classes like `"__proto__"` or `"constructor"`. Here, the vulnerability arises from the flow between the source `event` and the unvalidated function call on line 6. The `commHandler` function in Line 10 takes in a user input in the form of `event` variable, thus making `event` the source (highlighted in orange). This source variable now propagates taint across expressions in the program, as depicted by the dataflow edges 1-7 (in cyan). Specifically `event.data`, `data`, and `data.id` are all tainted expressions. Notice that the information flow happens across method boundaries through the `handlers["run"]` function in Line 1. In Line 3, the `handlers` object is dynamically indexed with the `data.id` tainted-variable and the indexed value is called as a function in Line 6 without checking whether the record `handlers` has a function with key `data.id` as its "own" property[1].

Figure 2b depicts a corresponding safe program with the transformation highlighted in green. The vulnerability is fixed by replacing the statement containing the sink, foo(data) with the if-statement (guard) between Lines 5 and 7. This guard blocks the execution of the sink on malicious user inputs. Specifically, `handlers.hasOwnProperty(data.id)` (Line 5) checks whether `data.id` is indeed an *own property* of the object and blocks the execution of the sink otherwise. Notice that the variables in the guard (`handlers` and `data.id`) do not syntactically appear in the sink statement (foo(data)). Hence, the repair strategy must use non-local dataflow information to repair.

Example 2. Figure 3a is an unsafe program with the cross-site scripting (XSS for short) vulnerability where untrusted user-input flows to an HTML response [3, 7]. A user can create a request where the `req.id` attribute contains malicious JavaScript code; this code will get executed on the application side making the program unsafe. The `requestListener` HTTP server handler on Line 1 reads the

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty

```
1  const requestListener = function ( req , res ) {
2    ...
3    var userId = req.id;
4
5    serveRequest(userId);
6    ...
7    var message = "Served user " + userId;
8    ...
9    res.send({"msg": message });
10 };
```

(a) Example for vulnerable program

```
1  const requestListener = function (req, res) {
2    ...
3    var userId = req.id;
4  + userId = escape(userId);
5    serveRequest(userId);
6    ...
7    var message = "Served user " + userId;
8    ...
9    res.send({"msg": message});
10 };
```

(b) Example for fixed safe program

Fig. 3. Example and corresponding fix for the XSS vulnerability

`userId` input in Line 3. This `userId` is used internally to handle the request as needed and then concatenated with a prefix into the `message` variable in Line 7. Finally, the tainted `message` variable is sent inside the HTTP response sink.

Figure 3b depicts a corresponding safe program with the transformation highlighted in green. The vulnerability is fixed by inserting a new statement containing the sanitizer. It removes the taint from a variable and makes execution of the sink on the variable safe. Specifically, the `escape` function in Line 4 sanitizes the `userId` variable and removes the taint. Note that this transformation is applied at an intermediate location between the source and the sink.

In this paper, we are interested in automatically repairing information flow vulnerabilities by introducing sanitizers and guards. We first use a snapshot of a training code base to learn repair strategies. Next, given an input program with a violation, we provide a high-level overview of how these learned strategies repair the input.

2.2 Applying repair strategies

Given an AST of an unsafe program annotated with sources, sinks, and vulnerable flows, our repair strategies follow a two-step process: find the *edit-location* and then apply an *edit-operation*, where:

(1) the edit-location is an AST-node where the edit occurs, and
(2) the edit-operation is a tree-edit operation at the edit-location. Since these vulnerabilities are fixed by introducing sanitizers or guards in programs, we support inserting a child and replacing a child with another tree as the edit operations.

Example 1. The repair in Figure 2 introduces an if-statement with `handlers.hasOwnProperty(data.id)` guard that makes the program safe. Thus the edit-location is the AST-node corresponding to the block statement between Lines 1 and 8. The edit-operation replaces the child containing the sink AST-node with the if-statement.

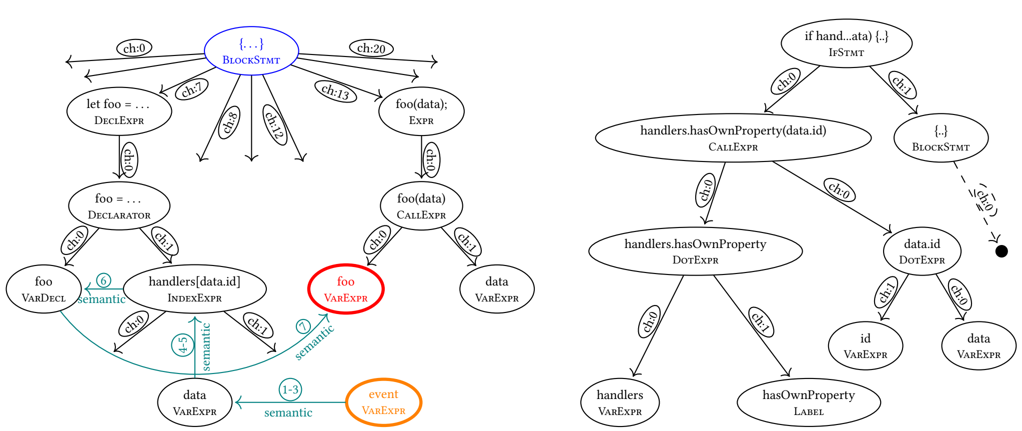
Example 2. The repair in Figure 3 inserts an assignment statement at Line 4 with the `escape` sanitizer to make the program safe. Thus the edit-location is the AST corresponding to the block statement between Lines 1 and 10. The edit-operation inserts the assignment statement as an additional child to this block.

The repair strategies have two components: (1) an edit localization component, which predicts the edit-location, and (2) an edit operation component, which constructs the input-program-specific edit that needs to be applied at the edit-location. We describe these components below.

Edit Localization. This component of the repair strategy encodes a path to the edit location starting from some known location in the program. In the case of information flow violations, the SA tool outputs the locations of the source and sink, and our repair strategy makes use of these anchors. An example encoding of reaching the edit location from the location of the source is as follows: *traverse data flow semantic edges in the annotated AST from the source along a path till a function call is encountered. Then, traverse syntactic AST edges from this call node to find the innermost block statement which is an ancestor to the function call. That block statement is the edit-location.* In Section 4 (Figure 9), we show a DSL for representing repair strategies. The above encoding is represented in this DSL using a KleeneEdge that allows crossing multiple edges of a particular *kind* until reaching a stopping-location. In Figure 2, this localization component follows the semantic edges 1-7 from event to reach the foo call in Line 6. Then, we traverse the syntactic parent of the `foo` call multiple times to reach the block statement of lines 1–8. Abstracting over the number of edges using KleeneEdge enables the localization component to generalize over other programs, which could be syntactic variations of this example. We learn these edge-types (syntactic or semantic) and stopping-locations (function call and block statement) from the training data.

Edit Operation. This component has abstract programs that are instantiated to ASTs using the input program. For example, a repair strategy can have an abstract guard `REF1.hasOwnProperty(REF2){REF3}`, where `REF1 REF2 REF3` are materialized with AST nodes that can be obtained by traversing *reference paths* of the input program. For Figure 2, instantiating this abstract guard with the unsafe program provides the guard `handlers.hasOwnProperty(data.id){foo(data)}`. Here, `REF2` (which materializes into `data.id`) is found by traversing the *reference path* containing `semantic-parent` edges from *semantic-location*[2] foo. Similarly, `REF1` and `REF3` materialize into `handlers` and `foo(data);` respectively by traversing reference paths associated with them.

3 DATA COLLECTION



(a) AST representation for the unsafe program in Figure 2a with the source in orange and the sink in red

(b) AST representation for the editprog corresponding to the unsafe program in Figure 2a

Fig. 4. Example of AST and AST of editprog

[2]Semantic Location is the node on the edit-path at the boundary between semantic-edges and syntactic-edges defined in Section 4

In this section, we describe how we collect examples for learning repair strategies without any version-controlled data. Specifically, we first detect witnessing-safe programs and corresponding witnesses using static-analysis witnessing (witnesses are sanitizers and guards that protect from vulnerabilities) in Section 3.1. Using these witness annotations, we generate unsafe programs and *edits* from the witnessing-safe program using a **witness-removal** step (Section 3.2). In the following, we define terminology for the AST data-structure we operate on.

AST refers to the abstract syntax tree representation of programs, augmented with data flow edges and annotations for sources, sinks, sanitizers, guards, witnesses etc. An AST is a five-tuple $\langle \mathcal{N}, \mathcal{V}, \mathcal{T}, \mathcal{E}, \mathcal{A} \rangle$, where:

(1) $\mathcal{N} = \{id_0, \dots id_n\}$ is a set of nodes, where $id_i \in \mathbb{N}$ for $0 \leq i \leq n$.
(2) $\mathcal{V}$ is a map from nodes to program snippets represented as strings. For a node n, we have that $\mathcal{V}(n)$ is a string representing the code snippet associated with n
(3) $\mathcal{T}$ is a map from nodes to their types defined by SA [2]. For example, CallExpr is the type of a node representing a function call, IndexExpr is the type of a node representing an array index, and BlockStmt is the type of a node representing a basic block of statements.
(4) $\mathcal{E}$ is a set of directed edges. Each edge is of the form (n_1, n_2, ET, z), where n_1 is a source node, n_2 is a target node, $ET \in \{\texttt{SynParent}, \texttt{SynChild}, \texttt{SemParent}, \texttt{SemChild}\}$ denotes the relationship from n_1 to n_2, as one of syntactic parent, syntactic child, semantic parent or semantic child, and $z \in \mathbb{Z}$ is the index of n_2 among $n_1's$ children if this edge is a child edge, and -1 if the edge is a parent edge.
(5) $\mathcal{A}$ is a set of annotations associated with each node. The annotations are from the set $\{\texttt{source}, \texttt{sink}, \texttt{sanitizer}, \texttt{guard}, \texttt{witness}\}$. We also refer to annotations using predicates or relations. For instance, for a node n, if an annotation `source` is present, we say that the predicate $\texttt{source}(n)$ is true.

A *traversal* or a *path* in an AST is a sequence of edges $e_0, \dots, e_{i-1}, e_i, \dots, e_k$ such that the target node of e_{i-1} is also the source node of e_i, for all $i \in \{1, \dots, k\}$. That is, e_{i-1} is of the form $(\_, n, \_, \_)$ and e_i is of the form $(n, \_, \_, \_, \_)$. The source node of e_0 is the source of this path and the target node of e_k is the target of the path.

Figure 4a depicts a partial AST corresponding to the unsafe program in Figure 2a. Each oval corresponds to an AST-node containing a type τ and an associated value. The dark edges denote the syntactic child edges. For example, the oval with value `foo(data)` is an AST-node with type CallExpr and has two children – `foo` and `data`, both with the type VarExpr. The semantic child edges are at the bottom in cyan. These edges correspond to the ones depicted in cyan in Figure 2a.

If P is an AST then we use P.`source` to denote the source node, P.`sink` to denote the sink node, and P.`witness` to denote the witness node. If the program has several sources, sinks and sanitizers then we generate a separate AST for each (`source`, `witness`, `sink`) triple. For a node n, its syntactic parent is n.`parent`, syntactic children are n.`children`, semantic parent is n.`semparent`, and semantic children are n.`semchildren`.

3.1 Static Analysis Witnessing

In this section, we show how to repurpose SA tools to generate witnesses. SA tools perform dataflow analysis to check for rule-violations in programs. They use pattern matching to identify known sources, sinks, sanitizers, and guards. For commercial tools, these patterns are implemented (and continuously updated) manually by developers and encode this domain knowledge. Next, SA checks if there exists a flow between a source and a sink that does not cross a sanitizer or guard. We capture this formally in Figure 5 (top two rules), and explain the notation used in it below.

$$\frac{\texttt{SemChild}(n_1, n_2) \qquad \neg\texttt{SanGuard}(n_1) \qquad \neg\texttt{SanGuard}(n_2)}{\texttt{SanGuardFree}(n_1, n_2)} \qquad \frac{\texttt{Source}(n_1) \qquad \texttt{Sink}(n_2) \qquad \texttt{SanGuardFree*}(n_1, n_2)}{\textbf{Vulnerability}(n_1, n_2)}$$

$$\frac{\texttt{SemChild*}(n_1, n_3) \qquad \texttt{SemChild*}(n_3, n_2) \qquad \texttt{SanGuard}(n_3)}{\texttt{SanGuardInMid}(n_1, n_3, n_2)} \qquad \frac{\texttt{Source}(n_1) \qquad \texttt{Sink}(n_2) \qquad \texttt{SanGuardInMid}(n_1, n_3, n_2)}{\textbf{Witness}(n_1, n_3, n_2)}$$

Fig. 5. Judgement rules for `Vulnerability` and `Witness` relations

SA tools encode domain knowledge about the vulnerability by annotating nodes as `Source`, `Sink`, `Sanitizer`, and `Guard`. So `Source`(n) is true iff the node n is a *source* node for a vulnerability. Next, SA tools perform dataflow analysis by defining the relation `SemChild`(n_1, n_2) which is true iff there is a datalow-edge between n_1 and n_2. Then the `Vulnerability`(n_1, n_2) relation can be defined as:

(1) n_1 and n_2 are source and sink nodes (`Source`(n_1) and `Sink`(n_2) are true)
(2) There exists a *path* between n_1 and n_2 which is free of sanitizers or guards (`SanGuardFree*`(n_1, n_2) is true). A path is free of sanitizers and guards iff every *edge* in the *path* is free of sanitizers and guards. An edge between n_1 and n_2 is considered free of sanitizers and guards (`SanGuardFree`(n_1, n_2) is true) iff $(n_1, n_2, \_, \texttt{SemChild}) \in \mathcal{E}$ and neither of n_1 or n_2 is a sanitizer or a guard

Here, we make the following observation - *this domain knowledge present in these annotations and relations is helpful beyond just detecting vulnerabilities.* For instance, simply using the sanitizer relation allows us to query the different kinds of sanitizers domain experts have specified. We use this observation to discover *witnessing-safe programs* i.e., programs having a source, sink, and a sanitizer or guard that *blocks* the information flow or, in simpler terms, make the program safe. In addition, we also detect the corresponding sanitizers or guards in the programs and refer to them as *witnesses* because they serve as the evidence of making the program safe. We call this procedure static-analysis witnessing (abbreviated as SA-witnessing). We define this as the `Witness` relation in Figure 5 (bottom two rules). Specifically, `Witness`(n_1, n_3, n_2) is defined as:

(1) n_1 and n_2 are source and sink nodes (`Source`(n_1) and `Sink`(n_2) are true)
(2) There exists a node n_3 such that it satisfies `SanGuardInMid`(n_1, n_3, n_2). `SanGuardInMid`(n_1, n_3, n_2) is true iff there exists a `SemChild` path between n_1, n_3, between n_3 and n_2, with the additional constraint of n_3 being a sanitizer or guard.

The difference between the `Vulnerability` relation (which SA populates) and `Witness` relations (which we want to find) is highlighted in red and green. Notice that while defining the `Witness` relation, we simply use the existing relations that define the `Vulnerability` relation. Thus, we argue that SA-witnessing can be implemented on top of SA by using the intermediate relations that SA is computing.

3.2 Witness Removal

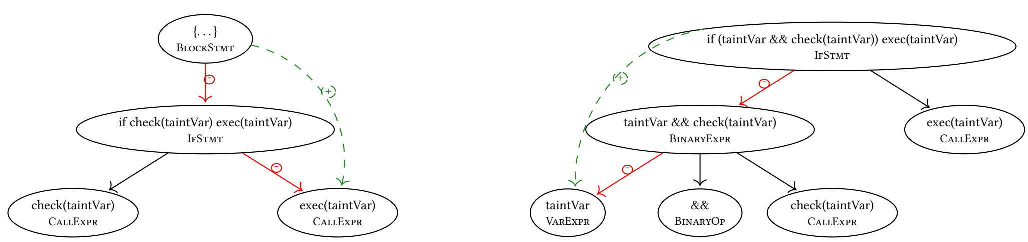
We obtain witnessing-safe programs and witnesses by applying SA-witnessing to a snapshot of a codebase. Recall that the witnesses block the flow between a source and a sink and thus help make programs *safe*. Hence, removing these witnesses will make the programs unsafe. Recall also that the witnesses are either sanitizing functions of the form `sanitize(taintedVar)` or guards of the form `if checkSafe(taintedVar) {executeSink(taintedVar)}`. We implement witness-removal perturbations that precisely remove the guard-checks and sanitizer-functions. Note that our goal here is to generate unsafe programs and corresponding edits that enable learning repair strategies that insert such witnesses. So, while we generate the unsafe programs by perturbation, they should look structurally similar to natural unsafe programs written by the developers, otherwise the repair

```
RemoveGuard (P_safe)
P_unsafe ← Copy(P_safe)
W ← P_unsafe.witness
W_par ← W.parent
W_grandpar ← W.parent.parent
if W_par.type = IfStmt :
    parindex ← GetChildIndex(W_grandpar, W_par)
    W_grandpar.children[parindex] ← W_par.children[1]
if W_par.type = BinaryExpr :
    parindex ← GetChildIndex(W_grandpar, W_par)
    witindex ← GetChildIndex(W_par, W)
    nonwitindex ← 2 - witindex
    W_grandpar.children[parindex] ← W_par.children[nonwitindex]
editprog ← W_par
editloc ← W_grandpar
out {P_unsafe, Edit(editprog,editloc), P_safe}
```

```
RemoveSanitizer (P_safe)
P_unsafe ← Copy(P_safe)
W ← P_unsafe).witness
W_par ← W.parent
if W_par.type = AssignExpr :
    parindex ← GetChildIndex(W_par, W)
    W_par.DeleteChild(parindex)
if W_par.type = Expr :
    unsanitized ← W.semparent
    parindex ← GetChildIndex(W_par, W)
    W_par.children[parindex] ← unsanitized
editprog ← W
editloc ← W_par
out {P_unsafe, Edit(editprog,editloc), P_safe}
```

Fig. 6. Sketch of RemoveGuard and RemoveSanitizer functions

strategies learned on this artificially generated data through perturbations would not generalize to code in the wild.



(a) RemoveGuard for guard inside a IfStmt

(b) RemoveGuard for guard inside a BinaryExpr

Fig. 7. RemoveGuard examples

We use RemoveSanitizer and RemoveGuard functions to programmatically remove the witnesses. A high-level sketch of these functions is illustrated in Figure 6. The functions use the structure of the corresponding AST (node types τ) to decide how to remove witnesses. Consider the RemoveGuard function. It first computes the parent (W_{par}) and grand-parent ($W_{grandpar}$) of the witness guard condition. Then if the type of W_{par} is IfStmt (i.e., program is of the form `if (witness) body` then we modify the AST edge from $W_{grandpar}$ and W_{par} to instead point to the body of the IfStmt (index 1 child is body of IfStmt). Similarly, if the type of W_{par} is BinaryExpr with operator `&&` (i.e. of the form `if (otherCond && guard)` or `if (guard && otherCond)`) then we again modify the edge from $W_{grandpar}$ and W_{par} to instead point to the non-guard child of BinaryExpr (`otherCond` in the example). Note that since BinaryExpr has 3 children, the index of non-guard child is index of guard-child subtracted from 2. Figure 7 depicts this removal on the AST level, where the syntactic edges in red are removed and the syntactic edges in green are inserted. In the end, the functions returns a tuple of the AST of the unsafe program (P_{unsafe}), AST of the safe program (P_{safe}) and an edit object (E) which stores

(1) AST for the removed witness (referred to as editprog)
(2) location in the AST where the witness is removed (referred to as editloc or L_E)

Since P_{unsafe} and edit-object can generate the safe program, we only propagate the unsafe programs and edits as the output of this step. Applying RemoveGuard function to the safe program

$$
\begin{aligned}
\textbf{@start}\ \textsc{Strategy}\ S &::= \texttt{Insert}(L, I, O)\ \mid\ \texttt{Replace}(L, I, O) \\
\textsc{GetTraversal}\ F &::= \texttt{GetEdge}(ET, I)\ \mid\ \texttt{GetKleeneStar}(ET, C) \\
\textsc{GetClauses}\ C &::= \texttt{GetClause}(\tau)\ \mid\ \texttt{GetNeighbourClause}(F, \tau)\ \mid\ C \wedge C \\
\textsc{GetIndex}\ I &::= \texttt{GetConstant}(z)\ \mid\ \texttt{GetOffsetIndex}(L, z) \\
\text{E-AST}\ O &::= \texttt{ConstantAST}(\tau, \mathit{value}, O_1, O_2, \cdots, O_c)\ \mid\ \texttt{ReferenceAST}(L) \\
\textsc{EdgeType}\ ET &::= \texttt{SynParent}\ \mid\ \texttt{SynChild}\ \mid\ \texttt{SemParent}\ \mid\ \texttt{SemChild} \\
\textbf{@input}\ \textsc{Program}\ P &::= \texttt{Input()}
\end{aligned}
$$

Fig. 8. Our DSL for representing repair strategies. Here, τ is the set of AST node types (Section 3), *value* is the set of possible string representations of AST.

in Figure 2b removes the IfStmt on Line 5 while preserving the `handlers[callerId](data);` statement and in fact produces the unsafe program in Figure 2a. Additionally, it returns the removed witness guard `if handlers.hasOwnProperty(data.id){ ... }` as the editprog and BlockStmt (blue oval in Figure 4a) as the edit location E.editloc. Figure 4b shows the AST for the editprog containing the IfStmt. The dashed line and dark circle correspond to the *removed* AST edge between the BlockStmt and the Expr `handlers[callerId](data)`.

Note that Figure 6 provides a high-level sketch of witness-removal and elides over implementation details that are required to make it work for real JavaScript programs. We discuss these issues in the implementation section (Section 5).

4 STRATEGIES AND LEARNING ALGORITHM

In this section, we describe how to learn repair strategies from the unsafe programs and edits collected in Section 3. We define a DSL (Section 4.1) to express repair strategies that take an AST of an unsafe program as input and generate a safe program as output. The DSL is expressive and can even express bad strategies that don't generalize well to programs in the wild. We provide examples of such bad strategies and good strategies that generalize well (Section 4.2). We learn good repair strategies in a data-driven manner using an example-based synthesis algorithm (Section 4.3).

4.1 DSL for repair strategies

We introduce a novel DSL to express repair strategies in Figure 8. At a high level, the strategies define a three-step process where they provide a computation to identify the edit-location node L_E, a computation to identify the child index I_E of L_E where repair happens, and a computation to generate the AST that must be placed at index I_E of L_E for the repair. The main part of these computations involve traversing paths of the input unsafe program P.

The top-level production rule of the DSL defines strategies, S, with type Strategy. GetTraversal, GetClauses, and GetIndex are all functions that take a Node n as input and return a Node, Bool, and Integer as output respectively. The edit-AST, E-AST, is similar to a syntactic variant of AST (i.e. no semantic edges) which we define in Section 3 with one addition. It has reference nodes that, when applying the strategy to the input AST of P, are materialized from sub-trees of this AST, where the root nodes of these sub-trees are identified by traversing paths in the input.

The strategy S is of two types, Insert or Replace. `Insert`(L, I, O) declaratively expresses the computation that computes the edit-location L_E by traversing the path supplied in L, then computes I_E, the index of edit-location, by evaluating $I(L_E)$, and inserts the materialization of O as a syntactic-child AST at index I_E of the edit-location L_E. `Replace`(L, I, O) is similar and performs a replacement instead of an insertion.

Node (L) is either the node corresponding to the source of vulnerability (P.source) or the target of the path corresponding to the traversal `ApplyTraversal`(L, F_k o F_{k-1} o $\cdots F_0$). Here, each F_i is a function that takes a node n as input, performs a traversal from n, and returns the traversal's target node n'. Thus, `ApplyTraversal` can be recursively defined as `ApplyTraversal`(F_0(L), F_k o F_{k-1} o $\cdots F_1$) if $k > 0$ and F_0(L) otherwise.

GetTraversal (F) defines a function that takes a node n and returns a node n' reachable from n and can be of two types. Given n, the `GetEdge`(ET, I) operator first finds the possible single-edge traversals of type ET and indexes it using I. Specifically, if edge type ET is a parent then it returns the parent of n. Otherwise, it finds a set of N of nodes that are connected with n via the edge type ET, i.e., $N = \mathcal{E}(n, ET)$, and returns the node $N[I(n)]$ at the index given by I. In contrast, `GetKleeneStar`(ET, C)(n) performs a KleeneStarTraversal that iteratively traverses edges of type ET, staring from input node n, until it reaches an edge whose target node n^i satisfies the condition defined by the clause C. Formally, KleeneStarTraversal can be defined recursively as $KE(n_1, ET, C) = C(n_1)?n_1 : (let\ t \in \mathcal{E}(n_1, ET)\ in\ KE(t, ET, C))$. Here, the node t, which is target of an edge with source n_1 and type ET, is chosen non-deterministically and our implementation resolves this non-determinism through a breadth-first search.

GetIndex (I) defines a function that takes a node n and returns a Integer. It is either a constant function that returns a fixed integer z or a `GetOffsetIndex`(L, z). `GetOffsetIndex`(L, z) takes a node n as input and returns an integer $DO(n, L) + z$, where $DO(n_1, n_2)$ returns the index of syntactic child of n_2 who is a syntactic ancestor of n_1.

E-AST (O) defines the edit AST with reference nodes which, given an input program P, are materialized to a concrete AST. The E-AST can either be a `ConstantAST` or a `ReferenceAST`. Specifically, `ConstantAST`(τ, *value*, O_1, O_2, $\cdots$, O_k) returns an E-AST that has a type τ, string representation *value*, and is recursively constructed with sub-trees $O_1 \cdots O_k$ as syntactic children, each of which can either be a `ConstantAST` or a `ReferenceAST`. The `ReferenceAST`(L), when applying the strategy, finds a node n in P by traversing the path described in L and returns a copy of the (syntactic) sub-tree of P rooted at n.

4.2 Example of strategies in our DSL

Figure 9 describes two possible repair strategies that are sufficient to repair the motivating example in Figure 2. We first describe the good strategy in Figure 9a, referred to as S_1, and then compare it with the bad strategy S_2 in Figure 9b.

Given the program P in Figure 2(a) as input, the strategy S_1 performs a replacement at index I of edit-location L_e with the materialization of O (line 20 of S_1). This process requires first finding the "semantic location" node L_S. To this end, the strategy first traverses a path from the node annotated as `source` by SA using `GetKleeneStar()` in Line 3 of S_1. This KleeneStarTraversal starts from `source`, traverses semantic dataflow edges, and stops at a node corresponding to an identifier being used as the function name in a function call. For the input program P, the traversal takes the semantic-child-edges 1-7 (Figure 4a) and stops at `foo` in Line 6 of Figure 2(a). Next, to reach the edit-location L_e, the strategy uses a KleeneStarTraversal that starts from L_S, traverses syntactic parent edges, and stops when it reaches a BlockStmt. For P, this traversal sets L_e as the node corresponding to the BlockStmt between Lines 1 and 8 of Figure 2(a). Next, in Line 7 of S_1, the index I is set to the index corresponding to the syntactic child of the edit-location L_e who is an ancestor of the semantic location L_s . For P, this index materializes into 13; the edge outgoing from blue BlockStmt in Figure 4a to an ancestor of semantic location (shown in red) has label `ch:13`. Next, we materialize the E-AST defined in Line 19 of S_1 by materializing the reference-nodes. The E-AST O serializes into `if (REF1.hasOwnProperty(REF2)) { REF3 }` where `REF1`, `REF2`, and `REF3` correspond to `ReferenceAST` operators with locations as L_{r1}, L_{r2}, and L_{r3}. L_{r2} traverses semantic-parent edges

```
1  P = input()
2
3  Ls = ApplyTraversal(P.source, GetKleeneStar("SemChild", GetClause("VarExpr") ∧ GetNeighbourClause("Parent", "CallExpr")) )
4  // Ls => foo;
5  Le = ApplyTraversal(Ls, GetKleeneStar("SynParent", GetClause("BlockStmt")) )
6  // Le => { ... }
7  I = GetOffsetIndex(Ls, 0)
8  // I => 13
9
10 Lr2 = ApplyTraversal(Ls, GetEdge("SemParent", GetConstant(0)) o GetEdge("SemParent", GetConstant(0)) )
11 // Lr2 => data.id
12 Lr1 = ApplyTraversal(Lr2, GetEdge("SynChild", GetConstant(0)) o GetEdge("SynParent", GetConstant(−1)))
13 // Lr1 => handlers
14 Lr3 = ApplyTraversal(Le, GetEdge("SynChild", I))
15 // Lr3 = foo(data);
16
17 O1 = ConstantAST("CallExpr", "...", ConstantAST("DotExpr", "...", ReferenceAST(Lr1), ReferenceAST(Lr2)))
18 O = ConstantAST("IfStmt", "...", O1, ReferenceAST(Lr3))
19 // O => if (handlers.hasOwnProperty(data.id)) { foo(data);}
20 S = Replace(Le, I, O)
```

(a) Example of a generalizing strategy

```
1  P = input()
2
3  Ls = ApplyTraversal(P.source, GetEdge("SemChild", GetConstant(0)) o GetEdge("SemChild" ... 7 times )
4  // Ls => foo;
5  Le = ApplyTraversal(Ls, GetEdge("SynParent", GetConstant(−1)) o GetEdge("SynParent" ... 3 times )
6  // Le => { ... }
7  I = GetConstant(13)
8
9  Lrexpr = ApplyTraversal(Le, GetEdge("SynChild", GetConstant(7)) )
10 // Lrexpr => let foo = handlers[data.id]
11 Lr1 = ApplyTraversal(Lrexpr, GetEdge("SynChild", GetConstant(0) o GetEdge("SynChild", GetConstant(1))
12 // Lr1 => handlers
13 Lr2 = ApplyTraversal(Lrexpr, GetEdge("SynChild", GetConstant(1) o GetEdge("SynChild", GetConstant(0))
14 // Lr2 => data.id
15 Lr3 = ApplyTraversal(Le, GetEdge("SynChild", I))
16 // Lr3 = foo(data);
17
18 O1 = ConstantAST("CallExpr", "...", ConstantAST("DotExpr", "...", ReferenceAST(Lr1), ReferenceAST(Lr2)))
19 O = ConstantAST("IfStmt", "...", O1, ReferenceAST(Lr3))
20 // O => if (handlers.hasOwnProperty(data.id)) { foo(data);}
21
22 S = Replace(Le, I, O)
```

(b) Example of a non-generalizing strategy

Fig. 9. Example repair-strategies in our DSL for the running example in Figure 2 with key differences highlighted in green and red. We also show the evaluations for the locations corresponding to the example as comments. The strategy on the top generalizes better because it uses semantic edges and KleeneTraversal.

from L_S (Line 10) and materialize into `data.id`. Similarly, L_{r1} and L_{r3} traverse syntactic children edges and materialize into `handlers` and `foo(data);` respectively. Thus, the E-AST O materializes into `if (handlers.hasOwnProperty(data.id)) { foo(data); }`, which is the required repair.

Now consider the repair strategy S_2 in Figure 9b. This strategy shares a similar structure with the earlier strategy but differs in the way traversals and the index I are computed. There are four key differences

(1) In order to reach L_S from P.source, S_2 performs the `EdgeTraversal` using semantic-child edge seven times in Line 3. The number of semantic edges varies widely across programs and prevents generalization to other scenarios. `KleeneStarTraversal` operator instead uses Clauses over nodes to find the edit-location.
(2) To reach L_e from L_S, S_2 performs the `EdgeTraversal` using syntactic-parent edge seven times in Line 5. Consider a program that instead assigns output of the function-call `let out = foo(data)`. S_2 will find AssignExpr as the edit-location and fail to generalize whereas S_1 will appropriately adjust and take four parent steps.
(3) In order to compute the index at which replacement needs to occur, S_2 uses a `ConstantIndex(13)` in Line 7 of Figure 9b, which effectively assumes that replacement should always occur at 13^{th} child of L_e and again doesn't generalize. S_1 on the other hand uses of `GetOffsetIndex` operator to instead compute index dynamically for a given input program
(4) In order to materialize reference nodes, S_2 uses syntactic edge traversals (Line 9 of Figure 9b) which assume definite structure about the structure of the program (`GetConstant(7)` used as syntactic child index to solve a long-ranged-dependency). S_1 instead uses semantic-parent edges to capture the semantics here and produces a better generalizing repair.

These programs highlight that our DSL is expressive enough to perform complicated non-local repairs in a generic manner. At the same time, while many strategies can repair a given program, all applicable strategies are not equally good. A key realization is that we *prefer shorter traversal functions* (KleeneStarTraversal over a long sequence of EdgeTraversal). Similarly, we *prefer the traversals with none or small constants*. For example, we prefer `GetOffsetIndex(`L_S `, 0)` over `GetConstant(13)` and semantic-parent traversal over syntactic-parent traversal with index `GetConstant(7)`. We use these insights to guide the search in our synthesis algorithm.

4.3 Synthesizing DSL strategies from examples

Given this high-level DSL, we will now describe our example-based synthesis algorithm. We take as input a set of unsafe programs and edits generated as output at the end of data collection step (Section 3). Let $\{(P_1, E_1), (P_2, E_2), \dots, (P_n, E_n)\}$. Here, P_i is the i^{th} unsafe program and E_i is the corresponding edit. Edit (E) contains the AST-node of the edit-location (E.loc), the *concrete* AST of the edit-program (E.editprog), and the type of edit i.e. Insert or Replace (E.type). We use these to learn high-level repair strategies in our DSL. Our goal is to combine specific paths, learned over examples that share similar repairs in different semantic and syntactic contexts, to obtain general strategies. Our repair strategies abstractly learn the following:

(1) Traversals for localizing edit-locations (L_E) and reference-locations (L_R). For example, `Ls` in Line 3 (Strategy S_1) depicts a `KleeneTraversal` abstraction we can learn from examples having a variable number of semantic-edges. Similarly, `I` in Line 7 (of S_1) depicts a generalized index expression we can learn from examples.
(2) E-AST which use reference-traversals. For example, `O` in Line 18 demonstrates templated-program-structure that we can learn from examples (say by generalizing from the witnessed guards `handlers.has(data)` and `events.storage.has(event.name)`).

We depict our synthesis algorithm in Figure 10. At a high-level, our synthesis algorithm, first pre-processes the inputs, storing the required *concrete* traversals. Next, it performs ranked pair-wise merging over the processed edits to synthesize strategies.

Pre-processing. In this step, given the programs and edits, we store the concrete traversals required for learning L_E and L_R (Line 4). Naively computing all such traversals is very expensive and also leads to *bad strategies*. Here, based on the insights from Section 4.2, we only compute the traversals that lead to shorter traversals which generalize better. In addition, we also share traversals between between L_E and L_R. Pre-processing has following three key steps:

(1) **Edit Traversals.** We compute the traversals between P.source and L_E (Line 12 of Figure 10) that have the form of a sequence of semantic-edges followed by a sequence of syntactic-edges. This allows abstracting variable-length sequences of semantic-edge traversals as a KleeneEdge (corresponding to an abstract KleeneTraversal). We implement this using BiDirecBFS method at Line 15. For every edit-traversal (T_e), we define *semantic-location* (L_S for brevity) as the last-node on the semantic (dataflow) traversal before traversing the syntactic-edges.

(2) **Compressing Edit Traversals.** We compress these edit-traversals using the Compress method in Line 23. It takes in a sequence of (syntactic or semantic) edges as input, greedily combines the consecutive edges with the same edge-type (ET) into a KleeneEdge. The KleeneEdge is constructed using the edge type ET, and a set of clauses C_i that satisfy the target node of KleeneEdge. These clauses are either $\lambda n.\mathcal{T}(n) = \tau$ that check the type or $\lambda n.\mathcal{T}(F_i(n)) = v$ that check the type of a neighbor. Compress returns a sequence of edges or KleeneEdges as output.

(3) **Reference Traversals.** For every node of the edit-program, we locate nodes in the AST with the same *value* using a LevelOrderBFS until a max-depth (Line 11). We perform this traversal from L_S (defined in (1) above). We thus share parts of traversals between locating L_E and L_R which helps in learning *better strategies*. The motivation behind using L_S is that the expressions necessary for repair will be close to L_S as it lies on the information-flow path.

Strategy Synthesis. Given the edits and the associated traversal meta-data, we synthesize the strategy by pair-wise merging (Line 4). MergeEdits, the top-level synthesis method, takes a pair of edits as inputs and returns a list of strategies satisfying the example edits. We synthesize the strategies recursively using a deductive search over the non-terminals of the DSL (Figure 8). Specifically, to synthesize an expression corresponding to a non-terminal, we deduce which production to use and recursively synthesize the non-terminals given by its production-rule. This has the following key components:

(1) MergeEdits: It takes pairs of edits as inputs and returns the strategy. It recursively synthesizes the traversal (for L_E), index, and E-AST. It combines and returns them using the edit-type.

(2) MergeTraversal: It takes two concrete traversals (sequence of edges or KleeneEdges) as inputs and returns the abstracted traversal. by merging elements in the sequence.

(3) MergeEdge: It takes two edges or KleeneEdges as inputs and returns a `GetKleeneTraversal` or `GetEdgeTraversal`. We combine two KleeneEdges using their edge-types and intersecting the clauses stored during pre-processing. We combine two edges using their edge-types, and recursively combining their indices.

(4) MergeIndex: It takes two integer indices as inputs and returns an abstracted index. If the two input indices are equal, we return a `GetConstant` operator with the input index value. Otherwise, we compute offset as the difference between input-index and index of child of n which has L_S as descendent (computed by $DO(n, L_S)$). We return this offset if they are equal and an empty-list otherwise.

(5) MergeProg: It takes two programs as input and returns a list of E-AST, where each list element can materialize into the input programs. If the top-level node in the programs have equal values and types, we combine them as a `ConstantAST`. Otherwise, we recursively combine

their children. Finally, we merge the reference-traversals corresponding to the input programs and combine them into `ReferenceAST`.

Our synthesis procedure is inspired by anti-unification [21] and we abstract the paths and edit-programs across different examples. Specifically, our `KleeneTraversal` and `OffsetIndex` functions allow generalization across paths having different number of edges and different indices where naive abstractions fail. Similarly, E-AST also resemble anti-unification over tree-edits. However, again we use traversals over syntactic and longer-context semantic-edges, for better generalizations and repairs.

Finally, note that while we perform pair-wise merges over the edits, the strategy synthesis algorithm can be extended to merge bigger cluster of edits together as well. However, from our experience, we find that the pair-wise merging performs well and is sufficient for our experiments.

5 IMPLEMENTATION

We use CodeQL [13] as our SA tool. It is an open-source tool where custom static analysis is implemented as queries in a high-level object-oriented extension of datalog. The queries follow a relational `select from where` syntax to query the program database. Thus, we are able to implement the **`Witness`** relation defined in Section 3 as queries in CodeQL.

We implement the witness-removal and strategy learning steps in C++. Specifically, we perturb the detected witnessing-safe programs using the AST structure of the programs as described in Figure 7. While implementing witness-removal, we need to handle two particular challenges

(1) **Ensuring naturalness of perturbed programs.** Consider the following program `{if (witness) {sink}}`. Here, during witness-removal, apart from removing the guard condition, we need to remove the additional braces around the `sink` as well. This is because the corresponding perturbed unsafe program generated (“`{{sink}}`”) would look unnatural and lead to non-generalizing repair strategies. We take care of such corner cases to the best of our abilities and leave investigating a more-thorough witness-removal pipeline for future work.

(2) **Capturing generalizing edits.** Consider the program `if (!witness) return custommessage`. Here, witness-removal step removes the entire IfStmt (including the return statement). However, while capturing the edit, we ignore the return-value and store the edit-program as only `if (!witness) return`. This is because the return values, error handling, and error messages are very customized across different codebases and not learnable using a programmatic strategy. We make such design decisions to capture these kinds of *generalizing edits* and discuss the implications in Section 8.

6 EXPERIMENTS

We present an empirical study of the proposed StaticFixer approach for JavaScript vulnerabilities with CodeQL as the SA tool. In particular, we look at two vulnerabilities, UDC-MembershipCheck and XSS, prevalent in JavaScript repositories. The goal of our study is to investigate how well the strategies learned by StaticFixer generalize to code in the wild, and how our method fares in comparison with state-of-the-art techniques for repair.

6.1 Datasets

We work with two types of datasets — one dataset exclusively for training and the other exclusively for evaluation.[3]

For *training* (e.g., for learning strategies in StaticFixer), we use JavaScript programs in the repositories available on LGTM [5] that are witnessing-safe (discussed in Section 3). We form a

[3]Our datasets can be found at https://tinyurl.com/5n6sjhh7

```
LEARN (I ≡ {(P_1, E_1), (P_2, E_2), ..., (P_n, E_n)})
for (P, E) ∈ I :
    PREPROCESS(P, E)
for i, j ∈ RANKSIMILAR(I) :
    out MERGEEDITS(E_i, E_j)

PREPROCESS (P, E)
traversals ← GETEDITLOCTRAVERSALS(P, E)
E.traversals ← traversals
C ← E.editcode
for node ∈ C :
    node.refs ← empty dict
    for T_E ∈ traversals :
        refs ← MAXLEVELBFS(T_E.semLoc, node.value)
        node.refs[T_E.semLoc] ← refs

GETEDITLOCTRAVERSALS (P, E)
L_E ← E.editloc
source ← P.source
traversals ← BIDIRECBFS(source, L_E)
for T_E ∈ traversals :
    T_E.semLoc ← GETSEMANTICLOC(T_E)
    T_E ← COMPRESS(T_e)
out traversals

GETSEMANTICLOC (T_E ≡ {e_0, e_1, ..., e_{z-1}})
for e_i ∈ T_E :
    if e_i.type="SynChild" or e_i.type="SynParent" :
        out e_i.end
out e_{z-1}.end

COMPRESS (T_E ≡ {e_0, e_1, ..., e_{z-1}})
ret ← empty list
i ← 0
while i < z :
    if e_i.type = SynChild :
        ret.add(e_i)
        i ← i + 1
        continue
    for j ∈ {i, i + 1, ..., z − 1} :
        if e_j.type != e_i.type : break
    if j ≤ i + 1 :
        ret.add(e_i)
        i ← i + 1
    else
        if j = z − 1 :
            C ← GETCLAUSES(e_j.end)
        else
            C ← GETCLAUSES(e_j.start)
        ret.add(KLEENEEDGE(e_j.type, C))
        i ← j + 1
out ret

MERGEEDIT (E_i, E_j)
ret ← empty list
type ← E_i.type                                        ▷ INSERT/REPLACE
traversals ← MERGETRAVERSALS(E_i.traversals, E_j.traversals)
for T_E ∈ edittraversals :
    global semLocs ← T_E.semLocs
    global T_S ← T_E.semTraversal
    editProgs ← MERGEPROG(E_i.editprog, E_j.editprog)
    index ← MERGEINDEX(E_j.index, E_j.index, T_E.endNodes)
    for editProg ∈ editProgs :
        ret.add(TYPE(T_E, index, editProg))
out ret

MERGETRAVERSALS ({T^1_1, T^1_2, ..., T^1_n}, {T^2_1, T^2_2, ..., T^2_n})
ret ← empty list
for every T^1_i and T^2_j :
    ret.add(MERGETRAVERSAL(T^1_i, T^2_j))
out ret

MERGETRAVERSAL (T_i ≡ {e^i_0, ..., e^i_n}, T_i ≡ {e^j_0, ..., e^j_m})
if n ≠ m : out empty list
mergeEdges ← empty dict
for k ∈ {1, 2, ..., n} :
    mergeEdges[k] ← MERGEEDGE(e^i_k, e^j_k)
out CARTESIANPRODUCT(mergeEdges)

MERGEEDGE (e_i, e_j)
if e_i and e_j are KLEENEEDGE and e_i.type = e_j.type :
    out GetKleeneTraversal(e_i.type, e_i.C ∩ e_j.C)
if e_i and e_j are EDGE and e_i.type = e_j.type :
    index ← MERGEINDEX(e_i.index, e_j.index, [e_i.start, e_j.start])
    out GetEdgeTraversal(e_i.type, index)

MERGEINDEX (I_i, I_j, [n_i, n_j])
if I_i = I_j :
    out GetConstant(I_i)
offset_i = I_i − DO(n_i, semLocs_i)
offset_j = I_j − DO(n_j, semLocs_j)
if offset_i =offset_j :
    out GetOffsetIndex(T_S, offset_i)
out empty list

MERGEPROG (C_i, C_j)
ret ← empty list
if C_i.value = C_j.value and C_i.type = C_j.type :
    ret.add(ConstantAST(C_i))
if C_i.type = C_j.type :
    mergedChildren ← MERGEPROG(C_i.children, C_j.children)
    ret.add(ConstantAST(C_i.type, mergedChildren))
refs_i, refs_j ← C_i.refs[semLoc_i], C_j.refs[semLoc_j]
reftraversals ← MERGETRAVERSAL(refs_i, refs_j)
for T_R ∈ reftraversals :
    ret.add(Reference(T_R))
out ret
```

Fig. 10. Sketch of our strategy learning algorithm

dataset ("**PAIREDPROGRAMS**") of (safe, unsafe) programs, using our SA-witnessing technique and CODEQL queries on LGTM [5], for the two vulnerability classes UDC and XSS. This dataset contains 800 paired programs for UDC and 66 paired programs for XSS; each pair consists of (i) a JAVASCRIPT witnessing-safe program that has a *witness* relation (a sanitizer or a guard, with the corresponding source and sink nodes) discussed in Section 3.1, and (ii) the corresponding "unsafe" program that is obtained by removing the witness using techniques discussed in Section 3.2.

For *evaluation*, we consider JavaScript code in the repositories on LGTM [5] ("**CodeInTheWild**") that are flagged as vulnerable (XSS or UDC-MembershipCheck) by CodeQL. We purge all the duplicate files — e.g., common JavaScript libraries are part of multiple repositories. After de-duplication, we have 330 unsafe JavaScript files from 204 repositories for UDC-MembershipCheck, and 672 unsafe JavaScript files from 595 repositories for XSS.

6.2 Compared Techniques

The datasets we use are real world JavaScript files, and there is no prior work on repairing information flow vulnerabilities in JavaScript — so we cannot use prior work in static repair as baselines (we qualitatively compare against them in Section 7). For example, if we were to use Phoenix [15], the most closely related system to StaticFixer, as a baseline then we would need to (i) port its front-end to consume JavaScript programs instead of Java, and (ii) generalize it to handle the repairs we seek from the more localized repairs it performs, which is a huge engineering effort. Furthermore, Phoenix implementation is proprietary and not available for this purpose. However, fine-tuning *neural techniques* to repair information flow vulnerabilities in JavaScript is feasible and we use them as baselines. The engineering effort required here is tenable as the models are available for public use, there are no front-end issues as they consume program strings, and we only need to provide natural language descriptions and program examples.

Therefore, we compare StaticFixer with the following state-of-the-art neural techniques for code synthesis, adapted for repair:

(1) CodeT5Js — we fine-tune the state-of-the-art code synthesis model CodeT5 [38] on the training set (**PairedPrograms**), to synthesize fixed code given vulnerable code as input.
(2) Codex — we use few-shot learning on the OpenAI's Codex model [16], to synthesize fixed code given vulnerable code and a set of paired programs (from **PairedPrograms**) as input.

As discussed in Section 4, StaticFixer solves the problems of both localization and repair. However, given a source-code file, the neural baselines do not have the ability to localize the program statements which need to be transformed for repairing the vulnerabilities. So, we parse the CodeQL warnings and provide the functions identified by CodeQL in its results as part of the input to the neural models. We give more details below.

Implementation details: We implement CodeT5Js as follows. We use the `CodeT5-small` variant of the model made up of 60M parameters initialised with pre-trained weights. For each of the two vulnerability classes, we finetune the model on the corresponding **PairedPrograms** dataset to produce fixed code given vulnerable code. Given a vulnerable file, we use CodeQL to identify function where the sink is, and pass this function as input to the model. We then finetune the model to produce non-vulnerable code (i.e., the corresponding sink function from the safe program in the training data), given the (localized) vulnerable code. During evaluation, we use beam-search decoding with a beam size of 20 and a temperature of 1.0, and generate 20 candidate snippets for each input. This ensures a fair comparison to StaticFixer where multiple strategies are applied to a given vulnerable code to generate multiple candidates (for 95% of the files in **CodeInTheWild**, StaticFixer generates at most 17 candidate fixes).

For the Codex model, we use a subset of unsafe, safe program pairs chosen from the **PairedPrograms** dataset to assemble a "prompt" encoding a description of the task, followed by the actual "question" (i.e., the vulnerable code) to generate output. In the experiments, we construct a prompt of the form $\{d^v, (u_1^v, s_1^v), \ldots, (u_k^v, s_k^v), u_q^v\}$; where d^v is a natural language description of how the vulnerability v should be fixed (taken from the CodeQL documentation), u_i^v and s_i^v are unsafe (vulnerable) and the corresponding safe code snippets that demonstrate how the vulnerability should be fixed, and finally u_q^v which is the *localised* vulnerable code snippet that we want the model to fix. The vulnerability-fixing (u_i^v, s_i^v) examples are drawn from a list of manually-chosen fixes (sink functions, to be consistent with CodeT5Js model) from **PairedPrograms**; Codex has a limit of 8000 input tokens, so the number of such examples in the prompt is determined dynamically depending on the size of u_q^v. We use a temperature setting of 0.9 and generate top 20 candidate outputs ranked by their probability.

6.3 Metrics

We report the number of successful fixes — we count an unsafe (as flagged by CodeQL) code as fixed if the corresponding output code from a method passes the vulnerability check of CodeQL. Additionally, we manually inspect the generated code to check if there are any unintended changes introduced in the original code. We deem the candidate fixes unsuccessful if there are any unintended changes or if they have any syntactic errors.

6.4 Results

The number of successful fixes obtained via our method and the baselines on the **CodeInTheWild** dataset is reported in Table 2. Recall that, for all the methods, we use the unsafe, safe program pairs from the **PairedPrograms** dataset for training. It is clear from the results that StaticFixer is not only significantly better than the baselines relatively but is also highly accurate in an absolute scale. In particular, StaticFixer (a) generates a successful fix (out of the possibly multiple fixes generated) for nearly 94% of the (vulnerable) files in the UDC-MembershipCheck class and for nearly 92% of the files in the XSS class, and (b) significantly outperforms, by as much as 2.5x, both the state-of-the-art neural techniques in both the vulnerability classes.

Even though we provide additional context to help the neural models localize (such as functions in the CodeQL warnings), these models fundamentally do not try to encode or exploit the domain knowledge. Fine-tuning very large neural models typically needs thousands, if not more, of training examples to be able to generalize well. However, in the real world, it is extremely challenging to collect training data of such scale without significant human effort.

Consider the following function in **CodeInTheWild** that is flagged for UDC-MembershipCheck vulnerability in line 2:

```
1 router.get('/api/crawlers/:type/:username', async (ctx) => {
2   const ojFunc = crawlers[ctx.params.type]
3   if (!_.isFunction(ojFunc)) {
4     throw new Error('Crawler of the oj does not exist')
5   }
6   ctx.rest(await ojFunc(ctx.params.username))
7 })
```

When passed as input to CodeT5Js, we observe that the top candidate fixes it generates fall into two categories: (a) adding a redundant type check after line 5 such as `if (typeof ojFunc === 'function') { ojFunc = crawlers[ctx.params.type] }`, or (b) adding an incorrect membership check such as `if (ctx.params.username in crawlers)` at line 2. Even though (b) captures the structure of the desired fix, it is not semantically correct, i.e., it checks for the wrong member `username` instead of `type`. We find that many of the unsuccessful cases for the neural models have similar failure modes — it reflects the inability of the neural models to capture both the structural and semantic contexts needed to produce intended repairs, for real-world scenarios. StaticFixer, by design, takes into account the data flow information and the semantics and produces semantically correct fixes in a vast majority of cases. In particular, for this example, it generates `if (crawlers.hasOwnProperty(ctx.params.type)) {ojFunc = crawlers[ctx.params.type];}`, at line 2. We

Method	UDC-MembershipCheck	XSS
CodeT5Js	127 (38.48%)	541 (80.51%)
Codex	220 (66.67%)	219 (32.59%)
StaticFixer	**310** (93.94%)	**617** (91.82%)

Table 2. Number of successful fixes by various methods on the **CodeInTheWild** dataset, out of (i) 330 JavaScript files for UDC-MembershipCheck, and (ii) 672 JavaScript files for XSS. All the methods are trained on the **PairedPrograms** dataset.

also notice from Table 2 that Codex performs poorly on XSS vulnerability with the most common failure case being that it copies over the vulnerable input as the output.

The success of our method can be attributed to two factors: a) our DSL provides a rich space of strategies, b) our learning algorithm learns a diverse set of strategies (292 in total for UDC-MembershipCheck, and 28 for XSS) guided by the limited set of "perturbed" safe programs and the witnesses in the training set. This diversity of strategies helps generalize to programs in the wild, which may deviate significantly in size, structure, and semantics from those in the training dataset. Furthermore, we find that, on average, StaticFixer produces about 4 unique fixes for a given vulnerable code in the **CodeInTheWild** dataset, of which about 3 are successful. Thus, StaticFixer is not only successful on a vast majority of the test files, but also produces multiple, unique correct fixes. This can be especially helpful in practice, when factors besides correctness can determine the suitability of a fix.

7 RELATED WORK

Automatic program repair is an active area of research. We refer the reader to [18, 31] for broad survey. Below, we list and compare related works that use static analysis as well as other approaches.

Static Analysis Based Repair. Systems such as FootPatch [36] and SenX [19] utilize static-analysis-information to create repair strategies. Specifically, FootPatch reasons about semantic properties of programs and SenX determines safety properties being violated to generate the patches. Unlike these approaches, StaticFixer uses static-analysis information (namely witnesses) to generate a paired unsafe-safe dataset of programs and then uses program synthesis to learn repair strategies from the dataset. Prior work uses static analysis to detect bugs, and then use cross-commit-data from manual fixes to create paired unsafe-safe dataset of programs, and learn repair strategies. For instance, SpongeBugs [29] uses SonarCube [8] static analysis to find bugs, and cross-commit data to create paired dataset. Similarly, Avatar [23, 24] uses FindBugs [4] static analysis and cross-commit data. GetAFix [14] similarly mines general tree-edit-patterns from cross-commit data using anti-unification. However, these fix-templates or edit patterns are purely syntactic whereas our repair strategies use semantic knowledge (specifically data flow) of programs, which is necessary for fixing information flow vulnerabilities. The Phoenix tool [15] makes more use of semantic information, and is closest to our approach in terms of learning repair strategies. However, we use SA-witnessing to learn repair strategies from a single snapshot of codebases whereas all previous approaches including Phoenix run SA across all historical commits of the repository to get paired cross-commit data, which is inherently noisy. Getting clean paired data from commits for bug fixes is a difficult problem, and we avoid this problem entirely. Additionally, our DSL supports KleeneEdge based operators that allow learning general repairs across examples where data flow paths have variable lengths, which is not supported by Phoenix.

Other Automated Program Repair approaches. Blade [37] and Lifty [33] repair information-leaks in programs using type analysis. HyperGI [30] performs repair on information-leak bugs using test suites. Refazer [34, 44] learns program transformations from developer edits using program synthesis. VurLe [28] and Seader [45] both learn program repairs from examples. CDRep [27] proposes

```
app.get('/perform/action', function(req, res) {
  let action = actions[req.params.action];
  ...
+ if actions.hasOwnProperty(req.params.action){
    res.end(action(req.params.payload));
+ }
});
```

(a) Example for fixed safe program

```
app.get('/perform/action', function(req, res) {
  let action = actions[req.params.action];
+ if (!actions.hasOwnProperty(req.params.action)){
+     return;
+ }
  res.end(action(req.params.payload));
});
```

(b) Example for fixed safe program

Fig. 11. Examples of fixes where broader application context is required to predict *natural* fixes

an approach to repair speculative leaks from cryptographic code. However, the approach requires users to manually write repair templates for cryptographic APIs. BovInspector [17] implements guard templates for fixing buffer overflow in C programs. Automated program repair techniques have used mutations of buggy programs to pass test cases in a suite [20, 22, 25, 26, 40, 41]. More recently, machine learning-based techniques have also started to gain attention for performing repair [11, 42, 43]. Our neural baselines emulate the recent advancements in neural large language models for code-generation, repair, etc. [32, 39].

8 DISCUSSION

We propose a novel approach to use static analysis and a repository of correct programs that satisfy a property, to automatically learn strategies to repair programs that violate the property. We have implemented our approach in the StaticFixer system. We evaluate our approach by performing repairs on two specific JavaScript vulnerabilities (unvalidated dynamic call and cross-site scripting) and learn general repair strategies. These repair strategies are able to automatically repair over 90% of the violations of these properties we found in over 1000 files collected from open-source repositories.

Our approach has two known limitations that can be potentially addressed in future work. The first limitation is due to our current implementation architecture. While our AST implementation can trace data flows across method boundaries, our AST is limited to a single file. A better AST builder would allow StaticFixer to repair flow vulnerabilities that cross file boundaries. The second limitation is a conceptual one. In addition to introducing sanitizers and guards judiciously, repairing information flow violations may also require application-specific side-effect handling, which is beyond the scope of this paper. For example, in Figure 11a, the guard blocks the dynamic execution of the function call to avoid the vulnerability. However, real-world fixes would also require appropriate error handling for the "else" branch, such as sending a suitable error message. The repair shown in Figure 11b suffers from the same issue, where it terminates function execution via a `return` without any error message or returning an error value. We imagine a human-in-the-loop repair process where our StaticFixer suggests the repair witnesses and human reviewers judge the repairs and additionally handle context-specific side effects such as error handling. We also envision a neuro-symbolic program repair system where the broader application context is *predicted* by a neural model like Codex [16] as a future direction.

Though we have evaluated our approach on two specific instances information flow properties, our approach has the potential to repair many classes of information-flow vulnerabilities such as null-dereferencing [6], zip-slips [1], tainted-path [10], SQL and Code Injection.

REFERENCES

[1] [n. d.]. Arbitrary file write during zip extraction (Zip Slip). https://codeql.github.com/codeql-query-help/javascript/js-zipslip/. Accessed: Nov 10, 2022.

[2] [n. d.]. CodeQL AST Types. https://codeql.github.com/docs/codeql-language-guides/abstract-syntax-tree-classes-for-working-with-javascript-and-typescript-programs/. Accessed: Nov 10, 2022.
[3] [n. d.]. Cross-Site-Scripting. https://owasp.org/www-community/attacks/xss/. Accessed: Nov 10, 2022.
[4] [n. d.]. FindBugs Project. https://spotbugs.github.io/. Accessed: Nov 10, 2022.
[5] [n. d.]. LGTM. lgtm.com. Accessed: Oct 1, 2022.
[6] [n. d.]. Null Dereferencing. https://owasp.org/www-community/vulnerabilities/Null_Dereference. Accessed: Nov 10, 2022.
[7] [n. d.]. Reflected cross-site scripting. https://codeql.github.com/codeql-query-help/javascript/js-reflected-xss/. Accessed: Nov 10, 2022.
[8] [n. d.]. SonarQube. https://docs.sonarqube.org/latest/. Accessed: Nov 10, 2022.
[9] [n. d.]. SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection. Accessed: Nov 10, 2022.
[10] [n. d.]. Uncontrolled data used in path expression (Tainted Path). https://codeql.github.com/codeql-query-help/javascript/js-path-injection/. Accessed: Nov 10, 2022.
[11] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *NeurIPS*.
[12] Olivier Arteau. 2018. Prototype Pollution Attack in NodeJS Application. *North Sec* (2018).
[13] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2
[14] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (oct 2019), 27 pages. https://doi.org/10.1145/3360585
[15] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952
[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).
[17] F. Gao, L. Wang, and X. Li. 2016. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 786–791. https://doi.ieeecomputersociety.org/
[18] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 01 (jan 2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013
[19] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. 539–554. https://doi.org/10.1109/SP.2019.00071
[20] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 295–306.
[21] Temur Kutsia, Jordi Levy, and Mateu Villaret. 2011. Anti-Unification for Unranked Terms and Hedges. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia (LIPIcs, Vol. 10)*, Manfred Schmidt-Schauß (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 219–234. https://doi.org/10.4230/LIPIcs.RTA.2011.219
[22] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 3–13.
[23] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2884955
[24] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 456–467.

[25] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[26] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 298–312.

[27] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. Association for Computing Machinery, New York, NY, USA, 711–722. https://doi.org/10.1145/2897845.2897896

[28] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. 2017. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples. In *Computer Security – ESORICS 2017*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer International Publishing, Cham, 229–246.

[29] D. Marcilio, C. A. Furia, R. Bonifacio, and G. Pinto. 2019. Automatically Generating Fix Suggestions in Response to Static Code Analysis Warnings. In *2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, Los Alamitos, CA, USA, 34–44. https://doi.org/10.1109/SCAM.2019.00013

[30] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra Cohen, and Justyna Petke. 2021. HyperGI: Automated Detection and Repair of Information Flow Leakage.

[31] Martin Monperrus. 2020. The Living Review on Automated Program Repair. (Dec. 2020). https://hal.archives-ouvertes.fr/hal-01956501 working paper or preprint.

[32] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–18. https://doi.org/10.1109/SP46215.2023.00001

[33] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *Proc. ACM Program. Lang.* 4, ICFP, Article 105 (aug 2020), 30 pages. https://doi.org/10.1145/3408987

[34] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *ICSE 2017* (icse 2017 ed.).

[35] Reudismam Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering* (Joinville, Brazil) *(SBES '21)*. Association for Computing Machinery, New York, NY, USA, 74–83. https://doi.org/10.1145/3474624.3474650

[36] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[37] Marco Vassena, Craig Disselkoen, Klaus Gleissenthall, Sunjay Cauligi, Rami Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proceedings of the ACM on Programming Languages* 5 (01 2021), 1–30. https://doi.org/10.1145/3434330

[38] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv* abs/2109.00859 (2021).

[39] Chun Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical Program Repair in the Era of Large Pre-trained Language Models. *ArXiv* abs/2210.14179 (2022).

[40] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 660–670.

[41] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 416–426.

[42] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback.

[43] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *International Conference on Machine Learning (ICML)*.

[44] Yuhao Zhang, Yasharth Bajpai, Priyanshu Gupta, Ameya Ketkar, Miltiadis Allamanis, Titus Barik, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, Gustavo Soares, and Ashish Tiwari. 2022. Overwatch: Learning Patterns in Code Edit Sequences. *CoRR* abs/2207.12456 (2022). https://doi.org/10.48550/arXiv.2207.12456 arXiv:2207.12456

[45] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng (Daphne) Yao, and Na Meng. 2022. Example-Based Vulnerability Detection and Repair in Java Code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 190–201. https://doi.org/10.1145/3524610.3527895