

# EffFix: Efficient and Effective Repair of Pointer Manipulating Programs

YUNTONG ZHANG, National University of Singapore, Singapore

ANDREEA COSTEA, National University of Singapore, Singapore

RIDWAN SHARIFFDEEN, National University of Singapore, Singapore

DAVIN MCCALL, Oracle Labs Australia

ABHIK ROYCHOUDHURY, National University of Singapore

This work introduces a static analysis driven Automated Program Repair (APR) technique for fixing memory errors. APR tools typically rely on a given test-suite to guide the repair process. Apart from the need to provide test oracles, this also introduces the over-fitting problem. Static analysis based APR techniques bypass these issues only to introduce new ones, such as soundness, scalability, and generalizability. In this work, we show how even these issues can be overcome and *soundly* repair memory bugs *at scale* by leveraging static analysis (specifically Incorrectness Separation Logic – ISL) to guide repair. Our key insight is that the abstract domain used by static analysis to detect the bugs also contains key information to derive correct patches. Our proposed approach *learns* what a desirable patch is by inspecting how close a patch is to fixing the bug based on the feedback from ISL based static analysis (specifically the Pulse analyzer), and turning this information into a distribution of probabilities over context free grammars. This approach to repair is *generic* in that its learning strategy allows for finding patches without relying on the commonly used patch templates. Furthermore, to achieve efficient program repair, instead of focusing on heuristics for reducing the search space of patches, we make repair scalable by creating *classes of equivalent patches* according to the effect they have on the symbolic heap. We then conduct candidate patch validation only once per patch equivalence class. This allows us to efficiently discover quality repairs even in the presence of a large pool of patch candidates. We implement our approach in a tool called EffFix. Experimental evaluation of fixing real world memory errors in medium to large scale subjects like OpenSSL, Linux Kernel, swoole, shows the efficiency and effectiveness of EffFix— in terms of automatically producing repairs from large search spaces. In particular, EffFix has a fix ratio of 66% for memory leaks bugs and 83% for Null Pointer Dereferences for the considered dataset.

## ACM Reference Format:

Yuntong Zhang, Andreea Costea, Ridwan Shariffdeen, Davin McCall, and Abhik Roychoudhury. 2024. EffFix: Efficient and Effective Repair of Pointer Manipulating Programs. 1, 1 (June 2024), 22 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Despite decades of efforts put into avoiding or mitigating memory safety errors (which are errors in handling memory in native programming languages such as C), recent surveys show that this class of issues still accounts for two of the three most dangerous software weaknesses reported in 2021 [1]. For example, reports show that 60% of the high severity security vulnerabilities and millions of user-visible crashes in Android are due to incorrect memory handling, while Google announced that 70% of all security bugs in Chrome in 2020 are memory safety issues. Given the ever increasing

---

Authors' addresses: Yuntong Zhang, National University of Singapore, Singapore, Singapore; Andreea Costea, National University of Singapore, Singapore, Singapore; Ridwan Shariffdeen, National University of Singapore, Singapore, Singapore; Davin McCall, Oracle Labs Australia; Abhik Roychoudhury, National University of Singapore.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

reliance on software and its growing complexity, if left unattended, memory safety bugs in legacy code will continue to prevail and would negatively impact the user experience and trust in software. Therefore, providing the tools and technologies to fix such bugs in a timely and efficient manner is a critical endeavour. Yet, this is easier said than done.

**Approaches-to-APR.** Advances in automated program repair (APR) techniques [2] show promise in dealing with the problem of bug repair. These techniques predominantly use test cases as a specification of program correctness. However, providing only a loose specification, tests are rarely exhaustive, thus making such techniques prone to over-fitting to the test. Furthermore, the conventional generate-and-validate approach assumes the following sequence of steps for each patch candidate: select a patch from a pre-defined search space and validate it for correctness by running the patched program against the given test cases. Repeated for each candidate patch and given a sufficiently large search space, this process turns out to be quite expensive.

FootPatch [3] and SAVER [4], the state of the art techniques for repairing memory safety bugs, reduce the reliance on test suites for patch validation in favour of using the advances in static analysis to determine the correctness of patches. FootPatch demonstrates that this direction is a promising one, managing to generate fixes for large codebases. SAVER further increases the effectiveness of static-analysis based repair by designing a novel representation of the program called object flow graph which summarizes the program’s heap-related behavior using static analysis, and resulting in a methodology which generates only safe fixes. However, this is still not quite enough. On the one hand, FootPatch is shown to produce unsound repairs [4], where fixing a memory leak bug could potentially lead to double-free issues. On the other hand, SAVER’s reliance on its object flow graph makes it a sound tool, but it restricts its bug-fixing capabilities to only those identifiable by a specific heap access pattern. Consequently, SAVER cannot address bugs like Null Pointer Dereference, which do not conform to a specific pattern.

**Our-approach-to-APR.** In this paper, we present a scalable, sound and generic methodology to fix memory related bugs without the need of test cases, implemented in a tool called EffFix. Inspired by the state of the art in repairing memory errors, EffFix relies on existing static analysis tools that are designed to find a semantically rich class of memory bugs. Differently than existing methods, EffFix is a sound and generic repair engine which is not restricted to rigid repair patterns. EffFix replaces the conventional patch synthesis followed by test-based validation with a novel synthesis and validation technique which work in tandem towards discovering what a correct patch is. In doing so it efficiently navigates the search space of candidate patches, and results in high repairability with a generic synthesis engine. To achieve this we adapt the advances of Incorrectness Separation Logic (ISL) [5, 6] for precise bug finding to the problem of automatically repairing memory leaks and Null Pointer Dereference bugs.

In a nutshell, our approach relies on ISL to describe the semantic effect the patch has on the symbolic heap, and to choose correct patches. Since the search space might be quite large, we propose to categorise patches into *equivalence classes* based on their semantic effect, and subsequently only validate one representative patch per class. Furthermore, to increase the likelihood of producing mostly correct patches, the synthesis checks how close a patch is to fixing the bug, by checking the patch’s effect on the bug, and focuses on regions in the search space which have a high chance of producing plausible patches. In particular, we describe the entire space of solutions using a *probabilistic context free grammar* and learn which of its production rules are most likely to be involved in a plausible patch. This allows for a generic, yet efficient synthesis engine, which is not constrained by custom bug templates or specifications.

The contributions of this work are as follows:

- a *scalable* approach for static analysis driven repair; the approach partitions large search spaces into semantic effect based equivalence classes, enabling *efficient validation* and scalability;

```

90 VERIFY_PARAM *VERIFY_PARAM_new(void){
91     VERIFY_PARAM *param;
92     param = OPENSSL_malloc(sizeof(VERIFY_PARAM));
93     (+) if (!param)
94     (+)     return NULL;
95     memset(param, 0, sizeof(VERIFY_PARAM));
96     verify_param_zero(param);
97     return param;
98 }

```

Fig. 1. An NPD bug and its fix in OpenSSL [7]

```

static CK_RV proxy_list_slots (Proxy *py, ...) {
    ...
    slots = calloc (sizeof (CK_SLOT_ID), count);
    ...
    py->mappings = realloc (py->mappings, ...);
    (+) if (py->mappings == NULL) free(slots);
    return_val_if_fail (py->mappings != NULL,...);
    ...
}

```

Fig. 2. A memory leak bug and its fix in p11-kit [4]

- a *generic* APR engine based on static analysis which does not require bug specific templates or specifications to fix a given bug; instead it relies on the feedback from the analyser to understand what a bug and its correct patch are.
- an *effective* navigation of the solution space based on probabilistic context free grammars, which favours the production rules with higher chance of deriving a plausible patch;
- an *open source* tool, EffFix, which implements our approach to fix memory safety issues.

## 2 MOTIVATION AND OVERVIEW

We next highlight some of the key aspects of our approach to APR for Null Pointer Dereferences and memory leaks and support these choices by means of examples.

**The case for static analysis.** Consider the Null Pointer Dereference (NPD) in Fig. 1, a bug previously reported in OpenSSL. Under low memory, `OPENSSL_malloc` returns `NULL`, thus leading to a null pointer dereference during the call to `memset` which takes `param` as an argument. The issue here is that explicitly checking `param` to be a non-null value—as per the fix indicated by (+) in the considered snippet—is not a standard practice within this project since, unlike `OPENSSL_malloc`, most `malloc` wrappers in OpenSSL abort if the result is `NULL`. The reservations developers have in acknowledging and fixing such bugs is highlighted in the conversations the authors of a static analyser used at Meta had with the OpenSSL maintainers [5]. The memory leak in Fig. 2 happens only on a very specific program path influenced by the outcome of the call to `return_val_if_fail`. Attempting to resolve it on any other path could either not fix the memory leak entirely, or lead to a double free issue.

To uncover and fix difficult to detect pointer manipulating bugs, the bug detector should understand the semantic effect a statement may have on the heap even in exceptional cases. This is hardly possible by means of dynamic testing because of the non-deterministic nature of dynamically allocated data structures and the difficulty of tracking alias information, which explains why so many memory related errors in production remain uncovered or unfixed for many years. In contrast, it has been empirically shown that static analysis is capable of uncovering even such corner cases since static analyses generally quantify over all possible effects a program may have [5]. We leverage the advances in ISL, a logic tailored for proving the presence of memory bugs, to describe the semantic effects programs have on the heap, and to guide the repair process towards the correct patch, i.e. a patch removing the unwanted semantic effects.

We shall use the potential NPD in Fig. 1 as our running example. An ISL bug detector is able to infer that a call to `OPENSSL_malloc` may result in two different valid program states, one corresponding to an empty memory footprint when the allocation fails, and another one where the allocation succeeds with a footprint comprising a single memory cell abstracted by a symbolic variable  $X$ :

```
[emp] param = OPENSSL_malloc [ok : param ↦ nil]
[emp] param = OPENSSL_malloc [ok : param ↦ X * X ↦ _]
```

Informally, the above abstract states (simplified for brevity) read as follows: starting from an **empty** heap, the program may result in a valid state (indicated by the label **ok**) where the resulting pointer *points to* nil, or in a valid state where the param points to a symbolic heap location X that stores an unspecified value *\_*. The first state causes issues at the call to `memset` at line 95 (ignoring the fix) since it requires param to point to a valid memory location. This possible Null Pointer Dereference is captured by the abstract states after the call to `memset` as follows:

```
[param ↦ nil]      memset(...) [err : param ↦ nil]
[param ↦ X * X ↦ _] memset(...) [ok : param ↦ X * X ↦ 0]
```

Since there is no modification in the **erroneous** symbolic state other than the label which changed from **ok** to **err**, it seems difficult to automatically derive a fix by simply looking at the program's abstract state (the approach taken by FootPatch). That is why, instead of adopting the abstract-state driven template-based patch search [3] which restricts the classes of derivable patches, we opt for a generic synthesis based on context free grammars (CFG), and only use the abstract state for validation purposes. We seek to derive patches that always lead to valid abstract states, i.e. no memory safety bugs, while keeping the code's functionality unchanged.

**The case for equivalence classes.** The advantages of a CFG driven synthesis are clear, i.e. genericity and simple machinery, and so are its disadvantages, i.e. poor efficiency due to a large search space which makes validation expensive. We aim to keep the advantages of our approach, while striving for efficiency. To this purpose, as we gradually derive more patches, we refine the search space of patches into equivalence classes, i.e. patches with *indistinguishable* effects on the symbolic heap, and, by doing so, we need not validate every generated patch but only one representative patch per equivalence class.

Consider the patches in Fig. 3—patches that could be generated for the example in Fig. 1. Although there are small syntactic differences between them, semantically they are equivalent. This equivalence is made obvious by the representation of the semantic effects these patches have on the symbolic heap depicted below each patch. We simplified the view of the heap, from formulae in ISL to sets of disjoint symbolic memory locations; in particular we use the empty set  $\{\}$  to denote an empty memory footprint, the singleton  $\{X\}$  to denote a memory footprint comprising a single memory cell, and the implication  $\text{param} = \text{nil} \implies \text{ok} : \{\} \wedge \text{ret} = \text{nil}$  to denote the pair of path condition  $\text{param} = \text{nil}$  on the left hand side of the implication, and corresponding heap abstraction on the right hand side of the implication (*ret* is a dedicated keyword indicating the returned value). In this new notation, the specification of the buggy program in Fig. 1 looks as follows:

```
param = nil  $\implies$  err :  $\{\} \wedge \text{ret} = \text{nil}$ 
param  $\neq$  nil  $\implies$  ok :  $\{X\} \wedge \text{ret} = \text{param}$ 
```

It becomes evident that all the patches in Fig. 3 have the same effects on the symbolic heap, and we need only validate one of them to conclude the validity of all the others. The size of one such class may exponentially grow with the size of the symbolic heap and the number of existing aliases.

**The case for probabilistic context free grammars (PCFG).** Exploring a large search space of patches may yield with significant time spent on incorrect patches. Ideally, we would like to spend less time in exploring patches belonging to classes of incorrect patches, and instead focus in regions in the search space (in the form of CFG productions) which is more likely to produce correct patches. To do this we equip the CFG with probabilities which indicate the likelihood of a certain production rule to be fired in a correct patch. However, understanding what a correct patch is in the absence of a specification is tricky. We break the patch correctness criterion into three simple requirements, and show how the

<pre> 3 param = OPENSSL_malloc(...); 4 (+) if (!param) 5 (+) return NULL; 6 memset(param, 0, ...); </pre> <p> <math>param = nil \Rightarrow ok: \{\} \wedge ret = nil</math>  <math>param \neq nil \Rightarrow ok: \{X\} \wedge ret = param</math> </p>	<pre> 3 param = OPENSSL_malloc(...); 4 (+) if (!param) 5 (+) return param; 6 memset(param, 0, ...); </pre> <p> <math>param = nil \Rightarrow ok: \{\} \wedge ret = param</math>  <math>param \neq nil \Rightarrow ok: \{X\} \wedge ret = param</math> </p>	<pre> 3 param = OPENSSL_malloc(...); 4 (+) if (param == NULL) 5 (+) return param; 6 memset(param, 0, ...); </pre> <p> <math>param = nil \Rightarrow ok: \{\} \wedge ret = param</math>  <math>param \neq nil \Rightarrow ok: \{X\} \wedge ret = param</math> </p>
---	--	---

Fig. 3. Equivalent patches and their effects for the bug in Fig. 1

<pre> 3 param = OPENSSL_malloc(...); 4 (+) if (false) 5 (+) return NULL; 6 memset(param, 0, ...); </pre> <p> <math>param = nil \Rightarrow err: \{\} \wedge ret = param</math>  <math>param \neq nil \Rightarrow ok: \{X\} \wedge ret = param</math> </p> <p>(a)</p>	<pre> 3 param = OPENSSL_malloc(...); 4 (+) if (param != NULL) 5 (+) return NULL; 6 memset(param, 0, ...); </pre> <p> <math>param = nil \Rightarrow err: \{\} \wedge ret = nil</math>  <math>param \neq nil \Rightarrow ok: \{X\} \wedge ret = param</math> </p> <p>(b)</p>	<pre> 3 param = OPENSSL_malloc(...); 4 (+) param = app_malloc(...); 5 memset(param, 0, ...); </pre> <p> <math>param = nil \Rightarrow ok: \{\} \wedge ret = param</math>  <math>param \neq nil \Rightarrow err: \{X, Y\} \wedge ret = param</math> </p> <p>(c)</p>
--	--	--

Fig. 4. Non-solutions for the bug in Fig. 1

probabilities ascribed to the CFG change *according to how many of these requirements the generated patch respects*, or in other words, according to how *close* the patch is to fixing the bug.

To simplify our explanation, we refer to the diagram in Fig. 5 in which the circle labelled with (1) is the set of all possible patches, while the intersection of spaces (3) and (5) is the set of plausible patches - (3) is the set of patches which only affect the buggy path and (5) is the set of patches that fix the bug without changing the program's returned values (we informally call this *weak-functionality preservation*). (2) is the sub-space of patches which have an effect on the buggy path but do not necessarily fix the bug and, in addition, they might affect other paths too. (4) is the sub-space of patches that fix the bug, but may break functionality. Ideally, we would like to gradually bias the search spaces towards the intersection of sub-spaces (3) and (5).

The first and most obvious requirement is for the patch to actually fix the bug. For example, all patches in Fig. 3 and the one in Fig. 4c fix the bug, however, the patches in Fig. 4a and Fig. 4b are non-solutions since they change nothing on the buggy path and therefore NULL can still flow into memset. We reward the production rules used in generating the patches which fix the bug (they are all in space (4) of solutions), while offering no rewards for those in the incorrect patches since they have no effect on the buggy path, i.e. patches outside space (2). An if – then production rule was used in generating both plausible and incorrect patches. Choosing not to reward it in the incorrect patches, instead of, say, penalizing it, allows us to still explore the space of patches containing if – then with the reward obtained from the correct patches, albeit guarded by different conditional expressions.

A closer inspection of the code in Fig. 4c reveals that although it fixes the NPD, this patch is actually a non-solution: apart from fixing the bug it also changes the intended functionality of the program since it affects the case where param is not NULL and introduces a

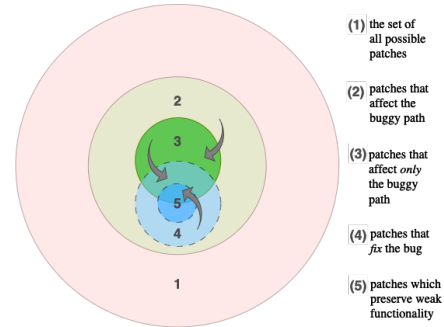


Fig. 5. Randomly navigate the space of patches (1) until a patch that affects the buggy path (2) is discovered. Continue the exploration in its vicinity until we hit a sub-space that only affects the buggy path (3), thus moving the exploration *closer* to discovering plausible patches. The sub-space of patches that fix the bug (4) is further refined into one with patches that preserve the code's functionality (5).

potential memory leak. This leads us to *the second requirement which*

*states that the patch should only affect the path on which the considered bug manifests*, e.g. when `param` is `NULL`, and *the third requirement which states that the patch should introduce no new bugs*; in other words, the patch in Fig. 4c is in the sub-space (4) in Fig. 5 since it fixes the bug, but not in sub-space (3) since it affects more than just the buggy path. Although a non-solution since it does not respect these two requirements, we still choose to reward the patch in Fig. 4c, albeit with a smaller reward than the patches in Fig. 3 receive. The reason for this design choice is that non-solutions may offer insights into how to remove the bug according to the sub-space they are in. For example, so far we have learnt that `if -` then is highly likely to be part of a correct patch, and that, although with a lesser probability, `app_malloc` can also fix the bug. This setup could potentially lead to a correct patch that wraps the `app_malloc` into a conditional affecting only the buggy path.

Generally, we choose to bias the search towards the space of plausible patches from two different but complementary dimensions which are evolving in parallel after a while: discovering the *correct path* and discovering the *correct effect*. To do that, the PCFG-based synthesis offers:

- no reward for the patches outside sub-space (2) since we learn nothing about a plausible fix from such patches - it is likely that at the beginning of the synthesis process most randomly generated patches will fall into this category;
- partial reward for path discovery for non-solutions in sub-space (2) but not in (3) - they offer information about the path on which we should look for a plausible patch;
- full reward for path discovery for patches in sub-space (3) - they only affect the buggy path;
- no reward for effect discovery for patches outside sub-space (4) - they do not fix the bug;
- partial reward for effect discovery for patches in (4) but not in (5) - they fix the bug but may change the program's behavior.
- full reward for effect discovery for patches in (5) - they fix the bug and preserve weak-functionality.

Sec. 3 formalises the proposed PCFG, and Sec. 5 discusses how we choose some of its parameters.

Getting back to the patch in Fig. 4c - a patch in sub-space (4) but not in (3) - we mentioned earlier that this patch receives a smaller reward than the patches in Fig. 3 receive. Yet, according to the guidelines on rewards we just mentioned it seems as if it should receive a full reward, hence seemingly contradicting our earlier statement. What happens is that the patch in Fig. 4c gets full reward for effect discovery, but gets only partial reward for path discovery, while the patches in Fig. 3 are fully rewarded for both path and effect discovery. In other words, the rewards on path and effect discovery compose leading to a smaller overall reward for the patch in Fig. 4c. This explains how non-solutions may still reveal useful information allowing the exploration of patches to get closer to plausible ones since generating solutions like those in Fig. 3 may be harder to come by as the space of plausible patches is often small, while useful non-solutions lie in larger spaces.

SAVER [4], the state of the art in repairing memory related bugs, is unable to generate a fix for our running example since the object flow analysis on which it operates manipulates events and non-allocation cannot be modelled as an event. FootPatch does handle null pointer dereferences but its search and template-based methodology cannot always generate fixes on specific paths, if the fix template has not been seen before - leading to restrictive fixes.

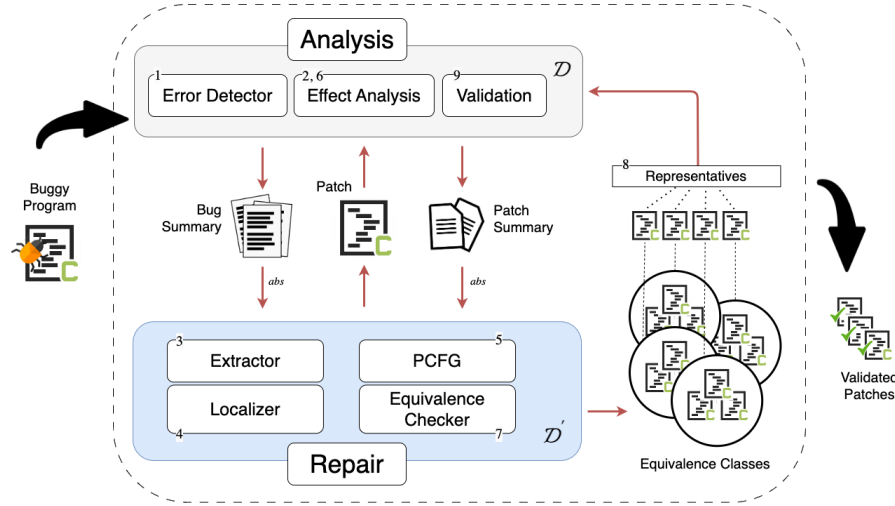


Fig. 6. Framework Overview

### 3 METHODOLOGY

This section describes how bugs are detected, how patches are synthesised using probabilistic context free grammars and subsequently classified into equivalence classes according to the effect they have on the footprint of the buggy program.

#### 3.1 Repair Framework Overview

Fig. 6 offers a summary view of our APR framework based on static analysis. In our approach, the static analysis is responsible for detecting the bug, for computing the semantic effects the bug and its corresponding patch have on the program’s memory footprint, and for validating the patch. To start with, the analysis defined on an abstract domain  $\mathcal{D}$  (ISL in our case) and taking a buggy program as input, detects the bug (1), computes the bug’s semantic effect (2) and it then creates a summary of the bug (the footprint of the buggy method, the path condition on which the bugs manifests, and the culprit statement). The bug’s summary is then used by the repair engine for extracting the ingredients for the patch (3) and for identifying the fix location (4). A patch is then synthesised using a PCFG (5). We investigate the effects the patch has on the memory footprint by creating a summary of the buggy method after having applied the newly created patch (6). Synthesised patches are then clustered into equivalence classes according to their effect on the symbolic heap (7). Only one representative patch per class (8) is then send out for validation (9). The results of the validation, e.g. does the representative patch remove the considered bug or does it affect other paths than the buggy ones, are transmitted back to the repair engine in order for it to fine-tune the probabilities ascribed to the PCFG. In other words, the probabilities implicitly reflect how the search space should be navigated. We note that, to break the dependency on the static analysis, that is, on the ISL domain, we further abstract the bug and patch summaries using a simplified abstract (meta-)domain,  $\mathcal{D}'$ , on which the repair functions. This meta-domain mostly retains information about what memory cells have been allocated and deallocated, and about the program paths and exit conditions.

### 3.2 Bug Detection

We build our approach on top of Pulse [8], an industry-grade static analysis tool which soundly detects memory safety violations. Pulse uses the latest advances in Incorrectness Separation Logic (ISL), a logic tailored to reason about the presence of bugs for heap-manipulating programs. Pulse first abstracts the C input program to an intermediate language, the Smallfoot Intermediate Language (SIL), and then runs an abstract interpretation engine to check for safety bugs.

**Program model.** A SIL core set of expressions and commands is depicted in Fig. 7. A program in SIL is a sequence of procedures, and a procedure is a composition of heap manipulating commands and standard commands, such as allocation, deallocation, conditionals, etc. The storage model comprises a stack and a heap, where the stack is a function from the set of program and logical variables to values, and the heap is a partial function from symbolic heap locations to values. A state thus models a stack and a heap, and together with an environment which tracks the values associated with program and logical variables it models a Pulse world.

**The abstract domain ( $\mathcal{D}$ ).** The abstract domain on which Pulse operates when symbolically executing the SIL commands is depicted in Fig. 8: a symbolic heap  $\Delta$  comprises a spatial term  $k$  and a pure, first order logical formula,  $\pi$  to account for pointer aliasing and non-heap information. The spatial term  $\text{emp}$  is an assertion to denote an empty heap,  $v \mapsto X$  is the points-to assertion for the program variable  $v$ , while  $Y \mapsto X$  is the points-to for logical variables.  $X \nrightarrow$  denotes memory deallocation, and the separation logic conjunction  $k * k$  denotes disjoint sub-heaps. An abstract state  $\Phi$  is defined as a pair of a program path  $\pi$  and a symbolic heap  $\Delta$ .

**Bug detection.** Pulse uses summaries (specifications) of predefined instructions to infer the summary (specification) of a given piece of code [5]. At the core of Pulse is the ISL (under-approximate) triple  $[\Phi_{pre}] c [\epsilon : \Phi_{post}]$  which asserts that any final state satisfying  $\Phi_{post}$  is reachable by executing  $c$  starting from an initial state satisfying  $\Phi_{pre}$ . Furthermore, the exit condition  $\epsilon$  indicates either a normal termination, i.e. `ok`, or a buggy one, i.e. `err`. The pair  $(\Phi_{pre}, \epsilon : \Phi_{post})$  describes the effect  $c$  has on one program path, and a set  $F$  (Fig. 8) of such effects describes the memory footprint of  $c$  where each effect in the set corresponds to a unique program path.

**Bug description.** A bug report in Pulse comprises the information  $\epsilon$  about the bug kind, e.g. null dereference, and the culprit statement  $c$ , e.g. the statement which dereferences a null pointer. On top, we record the summary of the method which contains the bug,  $F$ , and the path  $\pi$ , written as a first order logic formula, on which the bug manifests. A bug is defined in terms of the following tuple:

$$b ::= \langle \epsilon, \pi, c, F \rangle.$$

For a bug  $b$  we will often refer to an element of the tuple using the dot notation, e.g.  $b.F$ . The same notation is used throughout the paper for other kinds of tuples as well.

### 3.3 Patch Synthesis with Probabilistic CFGs

In this work, we employ a synthesis mechanism based on *probabilistic context free grammars (PCFG)* tailored for our approach to APR. In a CFG, a non-terminal symbol may be expanded in  $n$  different ways, e.g. a command  $c$  in Fig. 7 may be expanded in 8 different ways. In a PCFG, each production rule comes annotated with a probability  $p$ , denoting the probability of this rule being selected, with the proviso that the sum of probabilities of all  $n$  production rules should be 1, e.g.  $\sum_{i=1}^8 p_i = 1$  for the production rules of command  $c$ .

In our approach, instead of annotating the production rules with one probability, we do so with a pair of probabilities denoted by  $\langle p^\pi, p^e \rangle$ , with the same proviso holding separately for each probability in the pair, e.g.  $\sum_{i=1}^8 p_i^\pi = 1$  and  $\sum_{i=1}^8 p_i^e = 1$  for the production rules of command  $c$ . This design choice was made so as to be able to navigate the search



Variables $v$	Alpha-numeric identifiers $x, y$	Locations $loc$	Pointers $ptr ::= v \mid \text{NULL}$
Boolean Expression $b$	$::= \text{True} \mid \text{False} \mid b \vee b \mid b \wedge b \mid \neg b \mid x \text{ op}_r x \mid ptr \text{ op}_p ptr$		
Relational Operator $op_r$	$::= < \mid <= \mid == \mid != \mid > \mid >=$		
Pointers Operator $op_p$	$::= == \mid !=$		
Heap Manipulation $s$	$::= v := p \mid v := [ptr] \mid [ptr] := ptr \mid ptr = \text{new}() \mid \text{free}(v)$		
Commands $c$	$::= s \mid x := f(\bar{x}) \mid c; c \mid \text{ITE}(b, c, c) \mid \text{while}(b)\{c\} \mid \text{return } x \mid \text{return } ptr \mid \text{goto label}$		
Patch $P$	$::= \text{INSERT } c \text{ loc} \mid \text{COND False loc}$		

Fig. 7. Core (Simplified) Programming Language.

Exit condition $\epsilon$	$::= \{\text{ok}, \text{err}, \text{abort}\}$	Allocated symbolic heaps $H$	$\subseteq Loc$
Pure term $\pi$	$::= b$	Deallocated symbolic heaps $D$	$\subseteq Loc$
Spatial term $k$	$::= \text{emp} \mid v \mapsto X \mid Y \mapsto X \mid X \not\mapsto \mid k * k$	Aliases $A$	$\subseteq \mathbb{P}(\text{Vars} \times \text{Vars})$
Symbolic heap $\Delta$	$::= k \wedge \pi$	Meta-State $\phi$	$::= (\pi, \text{ret}, H, D, A)$
State $\Phi$	$::= \pi; \Delta \mid \exists X. \pi; \Delta$	Meta-Effect $e$	$::= (\phi_{\text{pre}}, \epsilon : \phi_{\text{post}})$
Effect $E$	$::= (\Phi, \epsilon : \Phi)$	Meta-Footprint $\mathcal{F}$	$::= \text{Set}(e)$
Footprint $F$	$::= \text{Set}(E)$		

Fig. 8. Abstract domain for bug detection ( $\mathcal{D}$ ).Fig. 9. Abstract domain for equivalence checking ( $\mathcal{D}'$ ).

space of patches from two different dimensions in parallel: finding patches with a high-probability of affecting only the path on which the bug was found (corresponding to probability  $p^\pi$ ), and finding patches with a high-probability of having an effect on the heap state which fixes the considered bug (corresponding to probability  $p^e$ ).

Assuming this pair of probabilities is set for each production rule (we detail in Sec. 3.4 how the probabilities are learnt), we generate patches by simply traversing the grammar and choosing production rules based on the product  $p^\pi * p^e$ , since we treat the event of generating a patch which affects the buggy path and the event of generating a patch with the correct memory effect to be independent of each other. To avoid the risk of leading to a very large (possibly infinite) parsing tree, we bound the size of the tree to a height  $h$ . However, this poses the risk of generating syntactically incorrect patches when the height  $h$  is reached. To avoid this, choosing the next production rule is a function of the rule's given probability and height: if the height of the generated tree is  $h$ , we prioritise production rules which lead to syntactically correct patches regardless of their probability; else, the probability is the sole deciding factor in choosing the next production rule.

### 3.4 Learning Probabilities

Starting from a PCFG with a uniform distribution (with regards to the pair of probabilities), we ascribe probabilities to this PCFG with the aim of increasing the likelihood of mostly navigating regions of plausible patches in the search space. Our strategy is to reward the production rules which lead to a patch that impacts the path the bug manifests on and those which lead to a patch that favourably affects the bug's memory footprint. This strategy allows us to learn the path the patch should affect even in the absence of a desirable effect on the memory footprint. And vice-versa.

Once a patch  $P$  has been generated from the PCFG and its effect derived, this effect is further examined to decide in which sub-space (as per Fig. 5)  $P$  belongs to. The outcome of this examination suggests how  $p^\pi$  and  $p^e$  of the production rules used to derive  $P$  should be adjusted, i.e. learnt. The adjustments can either be partial or full, as discussed in Section 2. For partial/full adjustment, our strategy uses pre-defined *adjustment factors* to determine how the probabilities are reassigned.

Let us assume a production rule  $r_1$  was used in deriving  $P$ , and that the current probability of  $r_1$  for path discovery is  $p_1^\pi = 0.4$ . Also assume that the grammar has only three rules  $c := r_1|r_2|r_3$ , and that the adjustment factors for partial/full adjustments are  $\alpha_p$  and  $\alpha_f$ , respectively. If the examination of the effect indicates that  $r_1$  should be given full reward for path discovery, then all probabilities for the rules in  $c$  should be adjusted. This adjustment involves re-distributing probabilities from  $r_2$  and  $r_3$  (the rules that were not used in deriving  $P$ ) to  $r_1$ , essentially rewarding production rules that were used to derive a correct/partially correct patch. Our strategy computes the probability of *not* choosing  $r_1$  and reassigns part of it proportional to  $\alpha_p$  to  $r_1$ , thus implicitly de-prioritizing  $r_2$  and  $r_3$ . Suppose  $p_2^\pi = 0.2$ , and  $p_3^\pi = 0.4$ . The probability of not choosing  $r_1$  is then  $0.2 + 0.4 = 0.6$ . The adjustment of the probabilities for path discovery are as follows:  $p_1^\pi$  is increased by  $0.6 * \alpha_p$ , while  $p_2^\pi$  is decreased by  $0.2 * \alpha_p$  and  $p_3^\pi$  is decreased by  $0.4 * \alpha_p$ .

This rewarding scheme gives a higher reward when the current probability of rule being rewarded is low, and gives a lower reward when the current probability becomes higher. This design choice makes the probability learning faster initially, and smooth down later on.

### 3.5 Patch Clustering

To reduce the cost of patch validation we progressively refine the solution space by identifying classes of *equivalent patches*, and proceed with only validating one representative patch per class. Two patches are equivalent if we can show that they lead to patched programs which have equivalent memory footprints, or, stated differently, they have the same effect when applied on the buggy program. Given an ISL triple  $[\Phi_{pre}] \text{ fnc } [\epsilon : \Phi_{post}]$ , the memory footprint of  $\text{fnc}$  is described by the two memory snapshots/states,  $\Phi_{pre}$  and  $\epsilon : \Phi_{post}$ , respectively. Reasoning about equivalent memory footprints would require reasoning about equivalent ISL formulas, which in turn requires ISL logic entailment checking. These requirements seem costly and highly dependent on the bug detector's domain. To break this dependency and make our approach agnostic to the bug detector, we design a meta abstraction on top of ISL which simplifies the description of the memory snapshot. Fig. 9 describes the meta domain  $\mathcal{D}'$  used for equivalence checking, while, defined as a recursive function *abs*, Fig. 10 introduces some of the main abstraction rules for translating a state from ISL to  $\mathcal{D}'$ . A memory snapshot in  $\mathcal{D}'$  is described by a tuple  $\phi$  comprising a path  $\pi$  in first order logic, a return value *ret* described in first order logic, a set of allocated symbolic memory cells  $H$ , a set of deallocated symbolic memory cells  $D$ , and a set of pointer aliases  $A$ . A meta-effect is a tuple  $e$  which comprises the exit condition  $\epsilon$  (ok or err) and two memory snapshots  $\phi_{pre}$  and  $\phi_{post}$ , corresponding to the inferred precondition and postcondition, respectively.

Considering the definition of a meta domain  $\mathcal{D}'$  as per Fig. 9, we can now define indistinguishable meta-effects in terms of indistinguishable states in this meta domain.

**Definition 1** (Indistinguishable meta-states). Two states  $\phi_1$  and  $\phi_2$  are said to be indistinguishable, denoted by  $\phi_1 \approx \phi_2$  if and only if the following condition holds:

$$\phi_1.\pi \Leftrightarrow \phi_2.\pi \wedge \phi_1.H = \phi_2.H \wedge \phi_1.D = \phi_2.D.$$

where the equality on sets is defined modulo the alias information in  $\phi_1.A$  and  $\phi_2.A$ , respectively.

$$\begin{array}{c}
\frac{\Delta \vdash \text{ret}}{\text{abs}(\pi; \Delta, \_) \triangleq \text{abs}(\Delta, (\pi, \text{ret}, \emptyset, \emptyset, \emptyset))} \quad \frac{}{\text{abs}(p_1 = p_2, (\pi, \text{ret}, H, D, A)) \triangleq (\pi, \text{ret}, H, D, A \cup \{(p_1, p_2)\})} \\
\frac{(\_, \text{ret}, H', D', A') := \text{abs}(k, (\pi, \text{ret}, H, D, A)) \quad (\_, \text{ret}, \_, \_, A'') := \text{abs}(\pi', (\pi, \text{ret}, H, D, A))}{\text{abs}(k \wedge \pi', (\pi, \text{ret}, H, D, A)) \triangleq (\pi, \text{ret}, H', D', A' \cup A'')} \\
\frac{(\pi, \text{ret}, H_1, D_1, A_1) := \text{abs}(k_1, (\pi, \text{ret}, H, D, A)) \quad (\pi, \text{ret}, H_2, D_2, A_2) := \text{abs}(k_2, (\pi, \text{ret}, H, D, A))}{\text{abs}(k_1 * k_2, (\pi, \text{ret}, H, D, A)) \triangleq (\pi, \text{ret}, H_1 \cup H_2, D_1 \cup D_2, A_1 \cup A_2)} \\
\frac{}{\text{abs}(Y \mapsto X, (\pi, \text{ret}, H, D, A)) \triangleq (\pi, \text{ret}, H \cup \{Y\}, D, A)} \quad \frac{}{\text{abs}(X \nrightarrow, (\pi, \text{ret}, H, D, A)) \triangleq (\pi, \text{ret}, H, D \cup \{Y\}, A)}
\end{array}$$

Fig. 10. Abstract domain transformation ( $\mathcal{D} \rightarrow \mathcal{D}'$ ).

**Definition 2** (Indistinguishable meta-effects). Two meta effects  $e_1$  and  $e_2$  are said to be indistinguishable, denoted by  $e_1 \approx e_2$ , if and only if the following condition holds:

$$e_1.\epsilon = e_2.\epsilon \wedge e_1.\text{pre} \approx e_2.\text{pre} \wedge e_1.\text{post} \approx e_2.\text{post}$$

So far we talked about a memory footprint as if it comprises a single pair of pre- and post-conditions. However, programs are often ascribed multiple such pairs to account for different behaviours on different program paths. A memory footprint is thus a disjunction of pair of states in ISL,  $F$  in Fig. 8, which corresponds to a set of effect tuples in the meta-domain  $\mathcal{D}'$ ,  $\mathcal{F}$  in Fig. 9. We define indistinguishable footprints as follows:

**Definition 3** (Indistinguishable meta-footprints). Two footprints  $\mathcal{F}_1$  and  $\mathcal{F}_2$  are said to be indistinguishable, denoted by  $\mathcal{F}_1 \approx \mathcal{F}_2$ , if and if the following condition holds:

$$\forall e_1 \in \mathcal{F}_1, \exists e_2 \in \mathcal{F}_2 : e_1 \approx e_2.$$

In other words, two footprints are indistinguishable if they have indistinguishable meta-effects on each path. Equivalent patches are now simply defined as:

**Definition 4** (Equivalent patches). Two patches  $P_1$  and  $P_2$  which lead to footprints  $\mathcal{F}_1$  and  $\mathcal{F}_2$ , respectively, when applied to the same buggy program, are said to be equivalent if and only if their corresponding footprint meta-abstractions,  $\mathcal{F}_1$  and  $\mathcal{F}_2$ , respectively, are indistinguishable:  $\mathcal{F}_1 \approx \mathcal{F}_2$ .

We use the above definition of equivalent patches to progressively partition the search space into classes of equivalent patches. The benefit of this partitioning is that we only need to validate one patch per class of plausible patches. Given a bug  $b$ , a class of plausible patches is one where all patches  $P$  meet the following condition:

$$\forall e \in P.\mathcal{F} : (e.\text{post}.\pi \Rightarrow b.\pi) \Rightarrow e.\epsilon = \text{ok}$$

In other words, the path on which the bug manifests is now labelled with an ok exit condition, i.e. the bug is fixed.

### 3.6 Patch Location and Ingredients

Pulse reports the location where the bug manifests, but we would like a fix at its source. For this purpose we adopt and further adapt the Spectrum Based Fault Localization or SBFL [9] to static analysis settings. SBFL requires test suite to generate pass/fail program traces. We collect this information from the program's specification which comprises both safe and buggy paths, thus feeding SBFL with a comprehensive "test suite" to cover all possible paths discovered by static analysis. The patch ingredients such as variables are computed by a simple taint analysis starting from the culprit object. Other ingredients such as constants and labels are collected within the same function scope as the fix location.

**Algorithm 1: MAIN**


---

```

1 Input: a buggy program  $\mathcal{P}$ 
2 Output: a map  $M$  from bugs to sets of patches
3  $M \leftarrow \text{InitMap}()$ 
4  $B = \text{detect the bugs in } \mathcal{P}$ 
5 for  $b \in B$  do
6    $\text{locs} \leftarrow \text{determine the fix location for } b$ 
7    $\mathcal{I} \leftarrow \text{collect vars and constants in } \mathcal{P} \text{ related to } b$ 
8    $G \leftarrow \text{a PCFG with terminals } \mathcal{I} \text{ and uniform distrib.}$ 
9    $C \leftarrow \emptyset$ 
10  for  $\text{loc} \in \text{locs}$  do
11    while  $P = \text{synthesise a patch using } G, \mathcal{I}, \text{loc}$  do
12       $C, G \leftarrow \text{REFINEEQUIVCLASSES}(C, P, G, b)$ 
13     $C' \leftarrow \text{filter } C \text{ for classes of plausible patches;}$ 
14     $C'' \leftarrow \text{validate } C' \text{ picking one patch per class;}$ 
15     $M \leftarrow \text{update } M \text{ with } b \rightarrow \text{rank}(C'')$ 

```

---

**Algorithm 2: REFINEEQUIVCLASSES**


---

```

1 Input: a set of existing patch clusters  $C$ , a patch  $P$ , a PCFG  $G$ , a bug  $b$ 
2 Output: updated patch clusters  $C$ , updated PCFG  $G$ 
3 for  $cls \in C$  do
4   if  $P.\mathcal{F} - b.\mathcal{F} == \text{summary}(cls)$  then
5      $C \leftarrow \text{add patch } P \text{ to the class } cls \text{ of } C$ 
6      $G \leftarrow \text{update } G \text{ according to } P \text{ and } cls$ 
7 if  $P \notin C$  then
8    $C, cls \leftarrow \text{add } P \text{ to a new class in } C$ 
9    $G \leftarrow \text{update } G \text{ according to } P \text{ and } cls$ 

```

---

**3.7 Putting it all together**

Now that we have identified most phases of our approach to APR, we outline how they are interconnected in algorithm 1. Given a buggy program  $\mathcal{P}$ , the algorithm incrementally populates a map  $M$  with classes of plausible patches for the bugs detected by Pulse (line 4). For each bug  $b$ , it determines all the possible locations where the patch could be inserted (lines 6) and collects the ingredients for the patch synthesis (line 7). Starting from a uniform distribution of a PCFG  $G$  (line 8), the synthesis of each new patch (line 11) triggers a refinement of the patch equivalence classes and an update of the probabilities (line 12). Lastly, we validate only the classes of plausible patches (lines 13-14) by choosing a representative patch per class - we use a simple ranking metric which measures the size of the patch's AST.

**Optimization.** We mentioned in Sec. 3.5 that two patches are equivalent if their footprints are indistinguishable. This implies that every time we generate a new patch we should test it against every other already generated patch to check whether they are indistinguishable, or in other words whether they belong to the say equivalence class. Although correct, this would be an expensive process. Instead, we expand on the definition of indistinguishable effects to define what a summary of an equivalence class is, and subsequently only compare a newly generated patch against equivalence class summaries.

This optimizations states that two patches are equivalent if they affect the buggy program in which a bug  $b$  manifests in the same way. To this purpose, we define a *distance relation* between a patch and a bug as the symmetric set difference between the sets of allocated and deallocated symbolic heaps for each effect in  $P$  and its corresponding effect in  $b$ :

$$P.\mathcal{F} - b.\mathcal{F} \triangleq \{e_P - e_b \mid e_P \in P.\mathcal{F} \text{ and } e_b \in b.\mathcal{F}\}$$

where  $e_P - e_b$ , the difference between effects, tracks how the exit condition changed,  $e_P.\epsilon \rightarrow e_b.\epsilon$ , the difference between pre-conditions, and the difference between post-conditions.  $P.\mathcal{F}$  and  $b.\mathcal{F}$  are the result of recursively applying the abstraction function  $abs$  on  $P.F$  and  $b.F$ , respectively. The difference between meta-states is defined as follows (where  $r$  is a flag that indicates whether the patch changes the returned value, or in other words whether weak functionality is preserved):

$$\begin{aligned} \phi - \phi_b \triangleq \{(\pi, r, H \ominus H_b, D \ominus D_b, A \cup A_b) \mid \pi \Rightarrow \pi_b \text{ and } r = (\text{ret} \Leftrightarrow \text{ret}_b) \\ (\pi, \text{ret}, H, D, A) = \phi \text{ and } (\pi_b, \text{ret}_b, H_b, D_b, A_b) = \phi_b\} \end{aligned}$$

It is this difference, namely  $P.\mathcal{F} - b.\mathcal{F}$ , that is used as equivalence class summary. With each new patch the equivalence classes are refined as depicted in algorithm 2, where the difference between meta-states is used to determine the patch equivalence (line 4). A benefit of refining the patch equivalence using this relation is that it allows us to compute the rewards for the PCFG (according to the case analysis described at the end of Sec. 2) at the equivalence class level, instead of computing them separately for each synthesised patch (line 6 and line 9).

## 4 IMPLEMENTATION.

We implemented our approach on top of Pulse<sup>1</sup>, a sound static analyser for bug finding in the Infer toolchain used at Meta. We use Pulse to detect bugs, to derive method summaries which we then use to inspect the effect patches have on the symbolic heap, and to validate patches. We use a number of custom CodeQL queries for collecting patch ingredients. For finding fix locations we use a bespoke instance of SBFL. For checking program path subsumptions we invoke CVC4, and for quantifier elimination when dealing with logical variables in path formulas we use Z3.

### 4.1 PCFG parameters.

**Patch size.** We mentioned in Sec. 3.3 that we limit the height of a patch tree to a constant  $h$ . Initial experiments on our dataset indicated that a tree height of at most 10 allows EffFix to discover patches for most of the considered subjects. A smaller height generally yields no results since patches would be larger than that, while a larger height entails a larger timeout due to the increase of the patch search space. We found the height limit of 10 to be a good compromise between efficacy and performance, and thus imposed this limit through out the evaluation.

**Adjustment factors.** In the current implementation we set the adjustment factors in probability learning to be 10 and 5 for partial and full rewards, respectively. The adjustment factors are chosen such that it takes a moderate number of continuous adjustments for the learning of probability to be evident. Starting from an initial low probability for a rule (e.g. 0.1), if it takes very few adjustments to reach a high probability (e.g. 0.9), the learning process would not be gradual. On the other hand, if it takes too many adjustments, the learning process might require an extended period to take effect. With such considerations, we set the adjustment factors to be 10 and 5 in the current implementation. These values allow a probability to increase from 0.1 to 0.9 in around 10 to 20 continuous adjustments.

<sup>1</sup>the version which comes shipped with Infer-7499c03

## 5 EVALUATION

To empirically validate the currently proposed static driven APR we have implemented our approach in a tool called EffFix. In our empirical study, we aim to answer the following research questions:

- **RQ1 (efficacy):** *How does EffFix perform against other similar tools?*
- **RQ2 (efficiency):** *How efficient are the equivalence classes in reducing the validation costs?*
- **RQ3 (effectiveness):** *How effective is the PCFG in navigating the search space of program patches?*

**Dataset.** We constructed our dataset of bugs to be fixed, by collecting (1) memory leak bugs from the benchmarks of SAVER [4], and (2) memory leaks and NPD bugs from OpenSSL in Pulse’s benchmark [5]. In other words, we consider in our evaluation those bugs that can be detected both by the versions of Infer used by SAVER and FootPatch, and by a more recent version of Infer<sup>2</sup> used by EffFix. Additionally, we collected some bugs found in the Linux kernel and added it to our dataset. SAVER and FootPatch rely on Separation Logic, a logic which over-approximates program states. This conservative approach may discover more bugs but it is prone to false positives, thus risking to put APR tools in the position of fixing non-bugs, e.g. fixing a false memory leak may lead to a double free. Instead, we built on Pulse’s Incorrectness Separation Logic, which under-approximates states, thus missing some bugs, but it guarantees EffFix only fixes true bugs. In total, there are 33 memory issues in our benchmark: 24 memory leaks and 9 NPDs. Tab. 1 contains a summary of these bugs, as well as the size of the subjects (43K - 17M lines of code) in which these bugs are witnessed.

**Setup.** Before conducting experiments on EffFix, we ran CodeQL and Pulse checker on each subject to generate static analysis database and bug detection reports, which serve as inputs to EffFix. Since the patch generation component in EffFix is probabilistic, we conducted all EffFix experiments with 10 repetition trials and reported the average across those 10 trials when appropriate. All experiments of EffFix and comparative tools was conducted using Cerberus [10] framework.

### 5.1 RQ1: Comparison with Other Tools

We compare the efficacy of EffFix against SAVER [4] and FootPatch [3], the state-of-the-art static analysis driven APR tools for memory bugs. We set a timeout of 20 minutes for EffFix and SAVER, since most developers prefer APR tools to produce repairs in under 30 minutes [11]. FootPatch was given a timeout of 1 hour because no patch was produced with the 20 minute timeout.

Tab. 1 summarizes the results of comparing EffFix to SAVER and to FootPatch, respectively. The two *#Bugs* columns indicate the number of bugs found by both EffFix’s underlying Pulse checker and the tool against which we compare. For example, EffFix’s Pulse finds 4 memory leaks for the `openssl-1` subject, but FootPatch only finds 1 which explains why we consider 4 bugs when comparing against SAVER and only 1 bug when comparing against FootPatch for the same subject. As a matter of fact even SAVER failed to detect the bugs in `openssl-X` and `LinuxKernel-v5.0`, but the authors of this paper managed to model their bug detection to account for these subjects as well, while the same modelling did not work for FootPatch.

Columns *Plausible* and *Correct* indicate the number of bugs for which each tool is able to find plausible and correct patches, respectively. A patch is plausible if it passes the analysis check, e.g. Pulse or Infer, and correct if it additionally passes manual inspection. The ground truth for the fixes in the `openssl-X` and `LinuxKernel-v5.0` subjects is provided in the form of developers’ fixes, by checking the commit history of the corresponding projects. Since there is no ground

<sup>2</sup>Infer-7499c03

Tab. 1. Comparison with static analysis based memory error repair tools in repairing C programs. Legend: kLoC: lines of code in the subject program (in thousands). #Bugs: total number of bugs being considered in the subject (we only consider bugs that can be detected by all tools). Plausible: number of bugs for which a tool can find plausible patches. Correct: number of bugs for which a tool can find correct patches.

Subject	kLoC	EffFix vs. Saver					EffFix vs. FootPatch				
		#Bugs	Plausible		Correct		#Bugs	Plausible		Correct	
			<u>EffFix</u>	<u>Saver</u>	<u>EffFix</u>	<u>Saver</u>		<u>EffFix</u>	<u>FootPatch</u>	<u>EffFix</u>	<u>FootPatch</u>
Memory Leaks											
Swoole (a4256e4)	43.0	3	2	2	2	2	3	2	2	2	1
p11-kit (ead7ara)	62.9	1	1	0	0	0	1	1	1	0	0
x264 (d4099dd)	73.2	6	6	6	4	3	6	6	0	4	0
Snort-2.9.13	320.8	8	6	8	4	8	8	6	0	4	0
OpenSSL-1.0.1h	279.2	4	4	0	3	0	1	1	0	1	0
LinuxKernel-v5.0	17184.7	2	2	2	0	0	0	0	0	0	0
Total		24	21	18	13	13	19	16	3	11	1
Null-Pointer Dereferences											
OpenSSL-1.0.1h	279.2	5	4	NA	3	NA	3	3	0	3	0
OpenSSL-3.0.0	480.86	3	2	NA	2	NA	3	2	0	2	0
LinuxKernel-v5.0	17184.7	1	1	NA	1	NA	0	0	0	0	0
Total		9	7	0	6	0	6	5	0	5	0

truth for the benchmark of SAVER, we solely rely on manual inspection to conclude the correctness of the generated patches for the subjects pertaining to this benchmark.

**Results.** For memory leaks, EffFix and SAVER have similar results. Given a total of 24 considered bugs, both tools found a correct patch for 13 bugs. In other words, the tools have each a fix ratio of 54%. When comparing against FootPatch on a total of 19 memory leaks, EffFix found correct fixes for 11 bugs out of 16 bugs for which it generated plausible patches, while FootPatch found 1 correct patch out of 3 with plausible patches. That leads to a fix ratio of 57% for EffFix in this context, and of 5% for FootPatch. For Null Pointer Dereferences, EffFix finds correct patches for 6 bugs out of the 9 considered in the comparison with SAVER, and for 5 bugs out of the 6 bugs considered in relation to FootPatch. That is a fix ratio of 66% and 83% for EffFix corresponding to the two considered evaluation contexts. SAVER is not applicable (NA) to NPDs since it uses pre-defined fix strategies. FootPatch, although it has capabilities to fix NPDs, it generated no plausible patches.

Fig. 11a captures the number of *unique* bugs each tool finds plausible patches for. EffFix found plausible patches for 11 unique bugs while SAVER found for 2 and FootPatch for 1. Fig. 11b depicts a similar diagram for correct patches, which shows EffFix finds correct patches for 10 unique bugs.

We note that although EffFix applies to NPD while SAVER does not, EffFix still correctly fixes 4 additional unique memory leaks compared to SAVER (out of the 10 unique bugs in Fig. 11b). For these 4 bugs, SAVER’s custom analysis either fails to analyse the bug report, or produces a patch with wrong path condition. On the other hand, SAVER generated a correct patch for 4 bugs for which EffFix did not. EffFix failed to generate a patch due to the large (automatically) constructed search space, which could have been alleviated by using a more strict selection criteria for patch ingredients or by increasing its timeout.

Compared to EffFix and SAVER, FootPatch found plausible/correct patches for fewer bugs. One possible reason is that FootPatch searches for candidate repair statements within the program, which could have two consequences. One is that it does not scale well for large codebases such as Snort and OpenSSL. In fact, FootPatch times out for these programs in our experiments. Another consequence is that it fails to find a patch which requires new expressions.

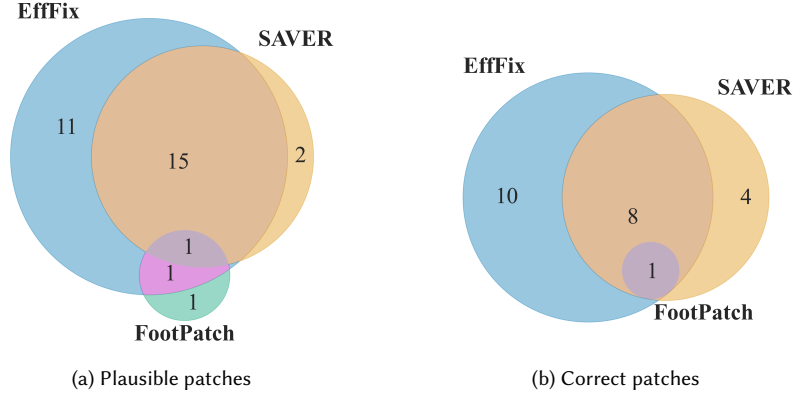


Fig. 11. Number of bugs for which each repair tool was able to generate a plausible/correct patch. The common bugs that multiple tools can generate a plausible/correct patch for are indicated by the overlaps.

**Answer to RQ1.** For memory leaks, the results of *EffFix* are similar or better than the state-of-the-art in repairing such bugs. Furthermore, owing to its generic patch synthesis engine, *EffFix* is also effective in fixing other kind of memory safety bugs such as Null Pointer Dereferences, where the outcomes indicate better overall results than the state-of-the-art.

## 5.2 RQ2: Efficiency of Patch Clustering

We evaluated *EffFix*'s strategy of clustering patches based on their effects. Tab. 2 details our results. We focus on the columns under *EffFix*, and postpone the discussion of those under *EffFix<sub>u</sub>* to Sec. 5.3. To counter for the randomness in the patch synthesis component, we conducted the experiments for ten trials and report the average results where appropriate. We used a 20-minute timeout for each run, which includes ingredients collection, patch synthesis and clustering. After the timeout, all patches that removed the targeted bug from the underlying Pulse analysis are considered as *plausible* (column #P<sub>p</sub>). Since all patches within one cluster are equivalent in the defined abstract domain, only one representative patch per cluster is selected as candidate for (manual) validation (the patch with smallest AST size is selected as representative). We refer to these patches as the *representative* plausible patches (column #P<sub>rp</sub>).

**Results.** Column #P<sub>p</sub> and #P<sub>rp</sub> highlight the effect of patch clustering. On average, *EffFix* generated 37.8 plausible patches for each bug, and, courtesy to patch clustering only an average of 5.2 patches are selected for validation purposes. In other words, patch clustering reduced the validation efforts by about  $\sim 7\times$  in our experiments, with the validation oracle being invoked 5.2 times on average for each bug instead of 37.8 times. The reduction in validation costs benefits not only the automated validation oracles such as static analyzers, but also the human developers who examine the plausible patches.

To give a complete picture to the reader, we discuss the bugs that could not be handled by *EffFix*. We note that *EffFix* did not generate plausible patches for 3 bugs (Bug 4, 5 and 10) in all trials. The main reason for not finding plausible patches within the timeout is likely the large search space. This larger search space is due to the relatively higher numbers of fix locations and other patch ingredients. Besides, *EffFix* did not work for Bug 20 because its bug trace spans multiple functions, which is not supported by our prototype implementation. Furthermore, *EffFix* also did not produce reliable results on Bug 29 because the program's abstract state hits the limit of disjuncts allowed by Pulse once the patches were applied to fix the buggy code. What this means is that although Pulse detects the bug in the original code, it is not guaranteed that it can still prove its presence after applying a possible incorrect patch if the patch increased the



Tab. 2. Details of EffFix (and EffFix<sub>u</sub>) in fixing memory errors. Legend: #Loc: number of different fix locations considered during repair. #PI<sub>p</sub>, #PI<sub>np</sub>, and #PI<sub>c</sub>: number of pointer variables, non-pointer variables and constants that are used as patch ingredients, respectively. #P<sub>s</sub>: count of syntactically different synthesized patches; #C: number of equivalence classes; #P<sub>p</sub>: count of plausible patches; #P<sub>rp</sub>: count of representative plausible patches. Mean denotes the arithmetic means across bugs.

ID	Subject	Type	#Loc	#PI <sub>p</sub>	#PI <sub>np</sub>	#PI <sub>c</sub>	EffFix				EffFix <sub>u</sub>			
							#P <sub>s</sub>	#C	#P <sub>p</sub>	#P <sub>rp</sub>	#P <sub>s</sub>	#C	#P <sub>p</sub>	#P <sub>rp</sub>
1	p11-kit	Leak	1	7	2	3	352	90	39.2	9.7	363	100	15.2	7.5
2	Snort	Leak	1	4	3	3	214	76	0.4	0.2	297	116	0.0	0.0
3		Leak	1	4	3	3	229	59	2.5	0.6	331	77	0.2	0.2
4		Leak	2	2	5	3	165	46	0.0	0.0	300	91	0.1	0.1
5		Leak	2	3	6	3	176	78	0.0	0.0	252	106	0.2	0.2
6		Leak	2	3	7	4	185	85	0.1	0.1	241	108	0.0	0.0
7		Leak	1	6	1	2	183	84	9.3	1.1	237	130	0.0	0.0
8		Leak	1	7	1	3	231	107	5.9	0.9	236	135	0.2	0.2
9		Leak	2	3	2	3	242	68	0.2	0.2	249	89	0.0	0.0
10	Swoole	Leak	2	5	8	6	100	43	0.0	0.0	98	44	0.0	0.0
11		Leak	2	2	3	3	273	77	57.4	13.0	272	70	20.3	8.6
12		Leak	2	3	1	3	372	82	66.6	15.2	411	67	26.0	10.2
13	x264	Leak	1	3	6	3	241	38	50.0	3.7	250	40	14.5	3.2
14		Leak	1	3	4	3	1102	215	131.5	26.0	1186	180	57.0	17.6
15		Leak	1	1	1	3	498	47	85.2	9.4	605	55	38.1	8.2
16		Leak	1	3	4	3	352	156	25.7	4.9	377	168	9.8	3.1
17		Leak	1	5	5	3	323	76	34.3	8.4	333	60	11.4	5.3
18		Leak	1	6	3	5	347	97	101.0	19.8	371	108	63.5	17.6
19	OpenSSL-1.0.1h	NPD	1	1	0	3	667	39	61.8	1.3	918	38	38.8	1.5
20		NPD	-	-	-	-	-	-	-	-	-	-	-	-
21		NPD	1	1	1	4	514	22	26.7	2.0	466	28	7.8	2.0
22		NPD	1	2	0	3	744	132	130.9	4.8	1141	212	78.7	5.9
23		NPD	1	5	1	2	173	86	7.7	1.8	199	125	2.3	1.6
24		Leak	1	1	0	4	700	14	216.5	4.0	1144	16	105.2	4.0
25		Leak	2	5	4	5	216	35	15.7	5.5	225	41	11.4	5.6
26		Leak	2	9	0	4	194	48	13.9	3.6	192	49	5.6	3.9
27		Leak	2	3	0	3	140	20	16.3	5.5	153	17	5.9	3.3
28	OpenSSL-3.0.0	NPD	1	1	0	2	406	45	36.4	2.2	592	44	24.4	2.1
29		NPD	-	-	-	-	-	-	-	-	-	-	-	-
30		NPD	1	1	1	4	376	37	24.2	1.8	425	43	5.8	1.2
31	Linux-5.0.0	NPD	1	1	2	3	829	194	26.8	6.0	1073	202	11.7	2.5
32		Leak	1	6	9	2	569	78	22.4	4.8	636	66	7.8	3.4
33		Leak	2	10	5	65	449	141	39.7	15.3	485	136	16.3	12.9
Mean				3.5	2.7	5.0	350	73	37.8	5.2	426	84	17.5	4.0

number of disjuncts above the limit which guarantees the soundness of the tool. Nevertheless, for other bugs which EffFix can find plausible patches for, patch clustering significantly reduces the validation effort. For example, for several bugs (e.g Bug 13, 19, 24, etc.), more than 90% of the plausible patches were identified as equivalent to the others, which means they can be excluded in future validation processes.

**Answer to RQ2.** Partitioning large search spaces into semantic effect based equivalence classes increases the efficiency of patch validation by up to 7x even for large scale codebases.

### 5.3 RQ3: Effectiveness of Probabilistic Grammar

We next investigate the effects of using a PCFG to navigate the search space. We performed an ablation study by disabling the probability learning in the PCFG. In other words, the same PCFG with a uniform probability distribution is used for both the patch synthesis and the clustering process. We refer to this version of our tool as EffFix<sub>u</sub> (with uniform probability distribution).

**Results.** The results of evaluating EffFix<sub>u</sub> are shown in Tab. 2, under the columns for EffFix<sub>u</sub>. Overall, the results also show that EffFix<sub>u</sub> finds lesser plausible patches on average, compared to EffFix (17.5 vs 37.8). The difference in numbers

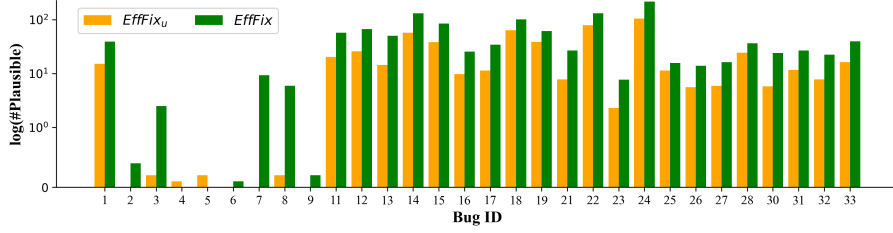


Fig. 12. Average number of plausible patches generated by EffFix and EffFix<sub>u</sub> for each bug, across 10 trials. Numbers are plotted in symmetric log scale. Bugs for which both tools found no plausible patches are excluded.

of plausible patches for individual bugs is also captured in Figure 12, which shows the numbers of plausible patches for each bug in log scale. Figure 12 shows that, for the bugs in the benchmark, EffFix consistently generated more plausible patches than EffFix<sub>u</sub>. This difference is likely due to the search bias: if the search is gradually guided towards regions of plausible patches by updating the PCFG, more plausible patches would be synthesized within the same time budget. Finding more plausible patches can also lead to more correct patches to be found. This is evident for the Snort subject, where EffFix<sub>u</sub> finds few or no plausible patches (for Bugs 2-9), and correct patches for 0/8 bugs. On the other hand, EffFix explores significantly more plausible patches, and finds correct patches for 4/8 bugs (as reflected in Tab. 1).

Furthermore, apart from exploring more plausible patches, EffFix also explores a higher number of plausible *regions*. On average, EffFix finds 5.2 plausible clusters while EffFix<sub>u</sub> finds 4. Nonetheless, EffFix<sub>u</sub> synthesized more patches on average (426 vs. 350) and created more clusters (84 vs. 73). This indicates that, although EffFix<sub>u</sub> explores more different regions in the search space, it explores more *implausible* regions compared to EffFix. EffFix, although synthesized less patches and explored fewer regions, was able to spend the time budget focusing on a larger number of plausible regions.

**Answer to RQ3.** *Augmenting the CFG with probabilities makes the navigation of the solution space more effective, guiding the search towards spaces more likely to contain plausible patches.*

## 5.4 Discussion

**Limitations.** A human oracle currently checks for plausible but incorrect patches which break functionality beyond changes in the heap’s shape. To ensure that we do not fix false positive bugs, we chose to build on top of Pulse since it has been shown to be sound with regards to bug finding, although incomplete. This takes care of the false positives concern specific to static analysis.

**Threats to Validity.** The benchmarks we chose for evaluation might not be representative for the classes of bugs we tackle, but we are constrained by the bugs discovered by both Pulse via EffFix, and by Infer via SAVER and FootPatch—the state-of-the-art in fixing memory errors for C/C++ programs. Furthermore, these APR tools are built for different categories of bugs, e.g. SAVER cannot handle NPD but fixes use-after-free/double-free, while FootPatch targets resource leaks too. Lastly, EffFix relies on the availability of the full source code involved in a memory safety issue. To mitigate this restriction, we modelled in Pulse the library calls whose source code is unavailable - we instructed Pulse with models for operations on string, such as `strlen`, `strdup`, `strcpy`, etc.

## 6 RELATED WORK

**Automated Program Repair.** Many program repair techniques have been studied in the last decade, largely for the purpose of fixing logical-errors. Recently the research community has studied fixing of security vulnerabilities [12, 13],

race conditions [14], students’ programming assignments [15], and so on. Program repair techniques can be classified into semantic repair [16, 17], search based repair [18, 19] or learning based repair [20, 21]. Search based repair techniques are known as generate and validate techniques, which heuristically search for a candidate patch in a space of program edits and validate to find a correct patch. Generally, validation is done using dynamic analysis with the aid of a test-suite. EffFix uses static analysis to validate the generated patches. Using a logic based semantic reasoning, EffFix provides additional evidence of correctness for the generated patches, thereby avoiding the patch over-fitting problem [22, 23] as well. Fixing memory errors has been studied previously using dynamic analysis [12, 13, 24], static analysis [3, 4, 25–27] and combination of both [28]. Dynamic approaches require a running test case as a witness for the memory error, and have been shown to be effective in fixing buffer-overflows [12, 13], NPD errors [24]. Our work is closely related to static analysis based repair of memory errors [3, 4]. FootPatch [3] generates patches for heap property violations detected using Infer [29]. Similarly, SAVER [4] generates safe patches for memory errors detected by Infer [29] and was shown to be scalable for larger programs. In both techniques, the patch generated is directly tied to the class of error reported by the static analyser. In contrast, EffFix uses a generalized grammar to synthesise patches of arbitrary types. Using a probabilistic grammar EffFix can dynamically adjust the probabilities to guide the search to correctly identify repair patterns, i.e. towards a suitable path condition or memory effect, which leads to finding more plausible patches. ProveNFix [30], developed at the same time with EffFix, is a static analysis tool grounded in temporal logic designed to detect violations of temporal properties at scale. The authors show how memory bugs can be formalised as violations of temporal properties, and enhance ProveNFix with repair capabilities to fix them. However, their approach is not directly comparable to ours since it is not fully automated thus adding considerable annotation burden on users which have to describe the bugs as violations of temporal properties and annotate the project accordingly.

A more recent line of works for APR is using the advances in generative AI to train models capable of repairing various classes of bugs through careful prompt engineering. For example, InferFix [31] uses a combination of a fine-tuned LLM for program repair and static analysis bug reports to detect and fix NPDs, resource leaks and thread safety violation bugs in C# and Java projects. While we are optimistic about the future of APR leveraging generative AI, we believe our work is complimentary and supports the continued advancements of non-AI approaches since a combination of the two is more likely to yield superior results in the future [32].

**Equivalence Classes.** Equivalence relations have been shown to benefit many search problems involving large search spaces such as mutation testing [33–35] and compiler testing [36, 37]. Recently, it was demonstrated to be effective for APR as well [38]. Equivalence relations can be used to explore larger patch spaces more efficiently. Value based test-equivalence used in [38], partitions the patch space based on runtime values observed during test executions. In contrast, EffFix defines an equivalence relation based on effect analysis.

**Probabilistic Grammar.** Augmenting probabilities with grammar production rules has been shown to be useful in program synthesis [39–41] and software fuzzing [42, 43]. Using a probabilistic grammar a software fuzzer can generate inputs based on production rule prioritization. In particular, previous work [43] has shown that evolving a probabilistic grammar can direct the search towards interesting inputs by favouring specific production rules. In contrast, EffFix uses a probabilistic grammar to generate program edits rather than program inputs. It evolves the probabilities to find a plausible patch by prioritizing the most promising production rules.

## 7 CONCLUDING REMARKS

We presented an automated program repair approach guided by static analysis. Our repair technique fixes null pointer dereferences and memory leaks. In our workflow, static analysis is used to both discover and fix a bug, thus alleviating

the classic over-fitting issue that test-based approaches normally suffer from. The novelty of our approach is two-fold. First, it is generic, requiring neither patch templates nor bug specifications. Instead, the repair engine incrementally *learns* what a correct patch may look like based on its effect on the symbolic heap. It stores this knowledge as a distribution of probabilities over a context-free grammar. Furthermore, we have empirically shown that the use of probabilistic context-free grammars leads to an effective patch space navigation. Second, to cope with the large search space of candidate patches, we proposed an efficient patch validation mechanism by clustering patches into equivalence classes according to the *effect* they have on the symbolic heap. The effect analysis on patches can be potentially extended to other use cases in the future, such as learning the effects of existing error handling routines in the program.

## 8 DATA AVAILABILITY

The artifact accompanying this paper is available from <https://doi.org/10.5281/zenodo.8389675>. It contains (i) the source code of EffFix tool, (ii) the dataset we used to evaluate it on, and (iii) scripts / data to reproduce our evaluation results.

## REFERENCES

- [1] “The 2021 common weakness enumeration top 25 most dangerous software weaknesses,” 2022, [https://cwe.mitre.org/top25/archive/2021/2021\\_cwe\\_top25.html](https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html).
- [2] C. Le Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, 2019.
- [3] R. van Tonder and C. Le Goues, “Static automated program repair for heap properties,” in *ICSE*. ACM, 2018, pp. 151–162.
- [4] S. Hong, J. Lee, J. Lee, and H. Oh, “SAVER: scalable, precise, and safe memory-error repair,” in *ICSE*. ACM, 2020, pp. 271–283.
- [5] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O’Hearn, “Finding real bugs in big programs with incorrectness logic,” *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: <https://doi.org/10.1145/3527325>
- [6] A. Raad, J. Berdine, H.-H. Dang, D. Dreyer, P. O’Hearn, and J. Villard, “Local reasoning about the presence of bugs: Incorrectness separation logic,” in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 225–252.
- [7] P. O’Hearn *et al.*, “Infer:Pulse-artifact,” <https://zenodo.org/records/6342311>.
- [8] —, “Infer:Pulse,” <https://fbinfer.com/docs/checker-pulse>.
- [9] R. Abreu, P. Zoetewij, and A. J. V. Gemund, “On the accuracy of spectrum-based fault localization,” in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION)*, 2007.
- [10] R. Shariffdeen, M. Mirchev, Y. Noller, and A. Roychoudhury, “Cerberus: A program repair framework,” in *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE ’23. IEEE Press, 2023, p. 73–77. [Online]. Available: <https://doi.org/10.1109/ICSE-Companion58688.2023.00028>
- [11] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, “Trust enhancement issues in program repair,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2228–2240.
- [12] Z. Huang, D. Lie, G. Tan, and T. Jaeger, “Using safety properties to generate vulnerability patches,” in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 539–554.
- [13] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, “Beyond tests: Program vulnerability repair via crash constraint extraction,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Feb. 2021. [Online]. Available: <https://doi.org/10.1145/3418461>
- [14] A. Costea, A. Tiwari, S. Chianasta, K. R. A. Roychoudhury, and I. Sergey, “Hippodrome: Data race repair using static analysis summaries,” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, mar 2023. [Online]. Available: <https://doi.org/10.1145/3546942>
- [15] M. R. Contractor and C. R. Rivero, “Improving program matching to automatically repair introductory programs,” in *Intelligent Tutoring Systems: 18th International Conference, ITS 2022, Bucharest, Romania, June 29 – July 1, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 323–335. [Online]. Available: [https://doi.org/10.1007/978-3-031-09680-8\\_30](https://doi.org/10.1007/978-3-031-09680-8_30)
- [16] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 691–701. [Online]. Available: <https://doi.org/10.1145/2884781.2884807>
- [17] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, “Concolic program repair,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 390–405. [Online]. Available: <https://doi.org/10.1145/3453483.3454051>
- [18] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan 2012.
- [19] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

- [20] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 298–312. [Online]. Available: <https://doi.org/10.1145/2837614.2837617>
- [21] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2021.
- [22] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 24–36. [Online]. Available: <https://doi.org/10.1145/2771783.2771791>
- [23] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? overfitting in automated program repair,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 532–543. [Online]. Available: <https://doi.org/10.1145/2786805.2786825>
- [24] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, “Dynamic patch generation for null pointer exceptions using metaprogramming,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 349–358.
- [25] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, “Safe memory-leak fixing for c programs,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. IEEE Press, 2015, p. 459–470.
- [26] J. Lee, S. Hong, and H. Oh, “Memfix: Static analysis-based repair of memory deallocation errors for c,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 95–106. [Online]. Available: <https://doi.org/10.1145/3236024.3236079>
- [27] —, “Npex: Repairing java null pointer exceptions without tests,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1532–1544.
- [28] H. Yan, Y. Sui, S. Chen, and J. Xue, “Automated memory leak fixing on value-flow slices for c programs,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1386–1393. [Online]. Available: <https://doi.org/10.1145/2851613.2851773>
- [29] C. Calcagno and D. Distefano, “Infer: An automatic program verifier for memory safety of c programs,” in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM’11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 459–465.
- [30] Y. SONG, X. GAO, W. LI, W.-N. CHIN, and A. ROYCHOUDHURY, “Provenfix: Temporal property guided program repair,” to appear 2024.
- [31] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, “Inferfix: End-to-end program repair with llms,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1646–1656. [Online]. Available: <https://doi.org/10.1145/3611643.3613892>
- [32] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. Tan, “Automated repair of programs from large language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1469–1481. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE48619.2023.00128>
- [33] R. Just, M. D. Ernst, and G. Fraser, “Efficient mutation analysis by propagating and partitioning infected execution states,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 315–326. [Online]. Available: <https://doi.org/10.1145/2610384.2610388>
- [34] Y.-S. Ma and S.-W. Kim, “Mutation testing cost reduction by clustering overlapped mutants,” *J. Syst. Softw.*, vol. 115, no. C, p. 18–30, may 2016. [Online]. Available: <https://doi.org/10.1016/j.jss.2016.01.007>
- [35] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, “Faster mutation analysis via equivalence modulo states,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 295–306. [Online]. Available: <https://doi.org/10.1145/3092703.3092714>
- [36] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 216–226. [Online]. Available: <https://doi.org/10.1145/2594291.2594334>
- [37] C. Sun, V. Le, and Z. Su, “Finding compiler bugs via live code mutation,” *SIGPLAN Not.*, vol. 51, no. 10, p. 849–863, oct 2016. [Online]. Available: <https://doi.org/10.1145/3022671.2984038>
- [38] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, “Test-equivalence analysis for automatic patch generation,” *TOSEM*, vol. 27, no. 4, pp. 15:1–15:37, 2018.
- [39] S. Bhaisaheb, S. Paliwal, R. Patil, M. Patwardhan, L. Vig, and G. Shroff, “Program synthesis for complex QA on charts via probabilistic grammar based filtered iterative back-translation,” in *Findings of the Association for Computational Linguistics: EACL 2023*. Dubrovnik, Croatia: Association for Computational Linguistics, May 2023, pp. 2456–2470. [Online]. Available: <https://aclanthology.org/2023.findings-eacl.189>
- [40] R. Ji, J. Liang, Y. Xiong, L. Zhang, and Z. Hu, “Question selection for interactive program synthesis,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1143–1158. [Online]. Available: <https://doi.org/10.1145/3385412.3386025>
- [41] Y. Xiong and B. Wang, “L2s: A framework for synthesizing the most probable program under a specification,” *TOSEM*, vol. 31, no. 3, 2022. [Online]. Available: <https://doi.org/10.1145/3487570>

- [42] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, “Evolutionary grammar-based fuzzing,” in *Search-Based Software Engineering*, A. Aleti and A. Panichella, Eds. Cham: Springer International Publishing, 2020, pp. 105–120.
- [43] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller, “Inputs from hell:,” *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1138–1153, 2022.