☯ Automated Program Repair from Fuzzing Perspective ☯

YoungJae Kim*
kyj1411@unist.ac.kr
UNIST
South Korea

Seungheon Han*
shhan@unist.ac.kr
UNIST
South Korea

Khamit Askar
khamit.askar@unist.ac.kr
UNIST
South Korea

Jooyong Yi
jooyong@unist.ac.kr
UNIST
South Korea

ABSTRACT

In this work, we present a novel approach that connects two closely-related topics: fuzzing and automated program repair (APR). The paper is divided into two parts. In the first part, we describe the similarities between fuzzing and APR both of which can be viewed as a search problem. In the second part, we introduce a new patch-scheduling algorithm called Casino, which is designed from a fuzzing perspective to enhance search efficiency. Our experiments demonstrate that Casino outperforms existing algorithms. We also promote open science by sharing SimAPR, a simulation tool that can be used to evaluate new patch-scheduling algorithms.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Automatic programming**.

KEYWORDS

Automated Program Repair, Fuzzing, Patch scheduling, Multi-Armed Bandit

ACM Reference Format:
YoungJae Kim, Seungheon Han, Khamit Askar, and Jooyong Yi. 2023. ☯ Automated Program Repair from Fuzzing Perspective ☯. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598101

1 INTRODUCTION

Software maintenance involves a continuous cycle of bug detection and fixing, which can be viewed as the yin and yang of software development. When a bug is detected, it represents the negative yin aspect, while bug-fixing activities represent the positive yang aspect of software maintenance. This cycle is essential for improving software quality, ensuring that software remains reliable.

As software quality becomes increasingly more important, research on bug finding and fixing is gaining momentum. Fuzzing has emerged as a popular and effective bug-finding technique and has been extensively studied recently. OSS-Fuzz [9] – a fuzzer developed at Google – alone has detected 8,900 vulnerabilities and 28,000 bugs as of February 2023. Fuzzing is now a routine practice at prominent vendors such as Google [2] and Microsoft [18].

As the bug detection capability improves, it is becoming unsustainable to fix all of the bugs manually. This is where automated program repair (APR) comes in. Since the seminal APR tool, GenProg, was introduced in 2009 [58], APR has been actively researched for the last decade or so. Using the current APR techniques, many bugs, if not all, can now be fixed automatically.

Fuzzing and APR, although seemingly opposite, share many similarities as they both solve a search problem. Fuzzing searches for bugs, while APR searches for patches to fix those bugs. In practical terms, the objective of fuzzing is to gather as many bugs as possible within a set timeframe. Similarly, APR can be leveraged to gather as many valid patches (i.e., patches passing all tests) as possible within a given time budget. In this paper, we will assume an APR approach that includes a patch ranking step where valid patches collected are ranked before being presented to the user. This approach is in line with recent works [10, 11, 15, 16, 26, 43, 60, 62, 70]. *In the first part of this paper, we will describe in more detail the relationship between fuzzing and APR.*

In the second part of this paper, we introduce a new patch-scheduling algorithm, which schedules the order of patches to be validated. The patch space is often too large to be checked exhaustively and this is why an efficient patch-scheduling algorithm is needed. Our approach is inspired by the principles of fuzzing, and we view the patch-scheduling problem through this lens. Specifically, we formulate the patch space as a tree, which we explore using stochastic tree traversal. The resultant algorithm, named Casino, is similar to the input-scheduling algorithm of grammar-aware mutation-based fuzzing. To facilitate efficient search, we employ multi-armed bandit algorithms to explore the structured patch space.

To assess the performance of our patch-scheduling algorithm, Casino, we compare it with the original scheduling algorithms implemented in six existing APR tools, including TBar [31] and AlphaRepair [62], the best-performing template-based and learning-based tool, respectively, at the time of writing. Additionally, we compare Casino with the state-of-the-art algorithm, SeAPR [6].

When assessed with the 395 buggy versions of the Defects4J [22] benchmark, Casino outperforms the original scheduling algorithms

*Both authors contributed equally to this research.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598101

across all six subject APR tools. When compared with SeAPR, Casino outperforms it in the majority of cases (4 out of 6 tools), while there is no clear winner between them in the remaining two. The high search efficiency of Casino also leads to high recall, enabling it to fix the largest number of bugs among all considered algorithms when given the same time budget.

In summary, we make the following contributions in this work:

(1) **New research directions.** We provide a new perspective on the relationship between fuzzing and APR and suggest new avenues for research.

(2) **Efficient algorithm.** We present Casino, a novel efficient patch-scheduling algorithm designed from a fuzzing perspective. Our experimental results show that Casino outperforms existing algorithms in terms of overall performance.

(3) **Experimental tool.** Finally, we share SimAPR, the simulation tool used in our experiments to evaluate various patch-scheduling algorithms. With SimAPR, assessing new patch-scheduling algorithms is made easy. SimAPR, along with the replication scripts, are available at:

> *https://github.com/UNIST-LOFT/SimAPR*

2 COMPARISON BETWEEN FUZZING AND AUTOMATED PROGRAM REPAIR

Both fuzzing and automated program repair (APR) have been intensively studied in the past decade [41, 73]. At a high level, these two techniques are similar in that they both solve search problems. Fuzzing searches for inputs that reveal bugs, such as crashes, while APR searches for valid patches that pass all available tests. Both problems are challenging due to the vastness of their search spaces. The search space of fuzzing includes all possible inputs to the program under test, while the search space of APR is the space of all possible patches. Since it is impractical to exhaustively navigate these huge search spaces, both fuzzing and APR conduct a search in the subset of the search space. The efficiency of finding bugs or patches depends on the scheduling algorithm employed, which selects the input or patch to explore in the next run. The following two subsections briefly describe the scheduling algorithms used in fuzzing and APR, respectively.

2.1 Fuzzing Scheduling Algorithms

Most fuzzers use stochastic scheduling algorithms that prioritize inputs that are more likely to expose bugs. To estimate the bug-revealing likelihood of an input, the program's past execution data is often analyzed. Black-box fuzzers prioritize inputs similar to those that previously uncovered bugs. Grey-box fuzzers extract more information from the execution data, such as code coverage, and use it to guide the search. The scheduling algorithms of fuzzing are typically online algorithms, meaning that the decision on which input to explore next is decided based on the execution data collected thus far. For more detailed and summarized descriptions of fuzzing scheduling algorithms, please refer to recent survey papers [36, 73].

Table 1: Comparison between fuzzing and APR (see § 2)

	Fuzzing	APR
Search space	Inputs	Patches
Search target	Bug-revealing inputs	Valid patches[†]
Oracle	Bug oracle (e.g., sanitizers)	Test suite
Schedule	Mostly stochastic	Mostly deterministic

[†]: A valid patch is also called a plausible patch in the literature of APR.

2.2 APR Scheduling Algorithms

Before initiating a patch scheduling algorithm, APR tools first run a fault localization technique to identify suspicious locations to modify. To limit the search space, most APR tools modify only a single program location, which we in this paper call APR^{SL}. Once the suspicious locations are identified, APR tools iterate over them from the most suspicious to the least suspicious. For each suspicious location, patches are generated using various techniques. Template-based techniques such as TBar [31] apply to the suspicious location predefined templates applicable at that location, while learning-based techniques such as AlphaRepair [62] use pre-trained machine learning models to generate patches at the suspicious program location. The generated patches are then validated using the given test suite. The above process is repeated to find valid patches (a.k.a., plausible patches) that pass all available tests.

Patch-scheduling algorithms typically employ one of two termination policies. The first policy is to halt the search once a valid patch is discovered, while the second policy, similar to fuzzing, is to continue searching for valid patches within a specified time limit. When the second policy is used, the discovered valid patches are ranked using a ranking method before being presented to the user. While the first policy was commonly used in the past, many recent APR tools [10, 11, 15, 16, 26, 43, 60, 62, 70] use the second policy because it can help alleviate the problem of the first-found valid patch being incorrect, even though it passes all tests. In this work, we focus on the recent APR approach that uses the second termination policy.

Notice that contrary to fuzzing, most APR^{SL} tools use deterministic scheduling algorithms. The scheduling order of patches is determined before the search starts based on the fault localization results, and the order is not changed during the search.

2.3 Research Opportunities

Table 1 summarizes the similarities and differences between fuzzing and APR discussed earlier. While these two techniques share similarities at a high level, they also exhibit distinct differences at a more granular level. This creates new avenues for research, including the following:

(1) Fuzzing efficiency has significantly improved recently, while most APR tools still rely on basic scheduling algorithms. Exploring the adaptation of techniques used to enhance fuzzing efficiency to APR could be a promising avenue for future research.

(2) The "grey-box" approach is proven to be highly effective in fuzzing, yet most APR tools still rely on either the "black-box"

or "white-box" approach. The white-box approach is utilized by the semantic-based approach [39, 42]. Exploring the integration of the grey-box approach into APR presents a promising research direction.

(3) Technology transfer can occur in both directions between fuzzing and APR. For instance, deep learning (DL) has been utilized in both fuzzing and APR, albeit in different ways. This presents numerous opportunities to combine and integrate diverse ideas.

In this work, we focus on the first item and propose a new efficient patch-scheduling algorithm.

3 FUZZING-INSPIRED PATCH SCHEDULING

In APR^{SL}, designing an efficient patch-scheduling algorithm boils down to the following questions: *how to select a location to modify and how to mutate the chosen location?*

How can a fuzzing perspective help us design a more efficient APR algorithm? *Our key observation is that the patch space of* APR^{SL} *has* ***a hierarchical tree structure****.* Consider the tree shown in Figure 1 – ignore the legend and $\texttt{s-score}_i$; they will be described later in § 3.1. The terminal nodes represent patches in the patch space. For each terminal node, say $\sigma_\star$, we have a path $\pi : r \to f_\star \to m_\star \to l_\star \to t_\star \to i_\star \to \sigma_\star$ where r represents the root node of the tree. In π, all non-terminal nodes between r and $\sigma_\star$ represent the structural component of patch $\sigma_\star$ such as file $f_\star$ and method $m_\star$. Applying patch $\sigma_\star$ is equivalent to traversing down π: (1) identifying the file $f_\star$, (2) locating the method $m_\star$, (3) zooming into the location $l_\star$, (4) using the template $t_\star$, and (5) finally choosing one of the patches obtainable by applying $t_\star$.

Now that we reformulate the patch space as a tree, we can explore the patch space via tree traversal, similar to *grammar-based fuzzing*. In grammar-based fuzzers like Langfuzz [19] and Superion [55], input space is formulated as an AST (abstract syntax tree) using the user-given input grammar. Then, these fuzzers perform mutation by randomly selecting a subtree of the AST and replacing it with another subtree extracted from other inputs in the seed pool.

Taking the perspective of fuzzing, we conceive the idea of *transferring grammar-based fuzzing to APR*. In this section, we formally define our structured patch space (§ 3.1) and present an overall idea about how to *navigate the patch space in a structured manner* (§ 3.2), which will be used as a backbone of our patch-scheduling algorithm. *Our patch-scheduling algorithm is stochastic* just like in fuzzing, which poses the traditional trade-off between exploration and exploitation. We *resolve this trade-off via multi-armed bandit (MAB) algorithms*, again inspired by fuzzing [56, 71]. We describe our patch-scheduling algorithm in § 3.4 after introducing in § 3.3 the two MAB algorithms we use. Lastly, we show how we incrementally construct the patch space on the fly while performing patch scheduling (§ 3.5). We call our patch-scheduling algorithm Casino.[1]

3.1 Structured Patch Space

Before we formally define the patch space used by Casino, we first define a patch configuration, a basic element of our patch space.

[1]Slot machines are colloquially called *one-armed bandits*.

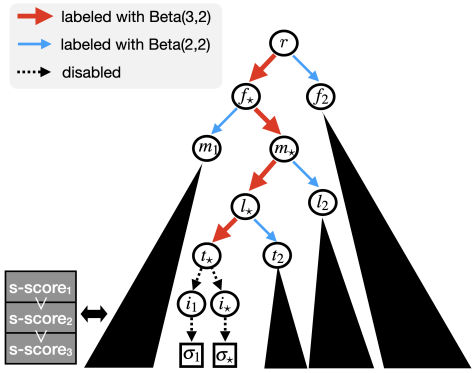


Figure 1: Patch-Space Tree (§ 3.1.1) and Lists (§ 3.1.2)

Definition 1 (Patch Configuration σ)**.** A patch configuration is a record $\sigma = \langle f : \texttt{file}, m : \texttt{method}, l : \texttt{loc}, t : \texttt{tmplt}, i : \texttt{idx}, c : \texttt{cvr} \rangle$ where its attributes denote the following.

- f: the file σ modifies.
- m: the method σ modifies; m is defined in file f.
- l: the location σ modifies; l belongs to method m.
- t: the template σ uses.
- i: the index of σ among the t-using patch instances at l.
- c: true if the patch was already covered; otherwise false.

The last field `cvr` is used to mark patches that are covered (selected) during the APR campaign. Initially, the `cvr` value is false for all patch configurations. We omit the `cvr` attribute from σ when we are not concerned with its value.

Notation. We use the notation $\sigma.a$ to access the attribute a of σ.

Terminology. Since any patch of an arbitrary APR^{SL} tool can be expressed as a patch configuration, we use *patch* and *patch configuration* interchangeably.

Definition 2 (Patch Space $\mathbb{S}[\![\tau]\!]$)**.** The patch space of τ, denoted with $\mathbb{S}[\![\tau]\!]$ where τ is an APR^{SL} tool, is defined as a set of patch configurations of τ.

We use two kinds of structures to represent the patch space: tree (§ 3.1.1) and list (§ 3.1.2).

3.1.1 Tree Structure. As mentioned, we view the patch space as a tree, as formally defined in the following:

Definition 3 (Patch-Space Tree $\mathcal{T}[\![\tau]\!]$)**.** Given a patch space $\mathbb{S}[\![\tau]\!]$, we reformulate it to a patch-space tree $\mathcal{T}[\![\tau]\!]$. We first define the nodes of $\mathcal{T}[\![\tau]\!]$ and subsequently, the edges of $\mathcal{T}[\![\tau]\!]$. To define *nodes*, we use the `level` function that takes as input the level of $\mathcal{T}[\![\tau]\!]$ and returns a set of nodes constituting that level:

- $\texttt{level}(0) = \{\text{the root node } r\}$
- $\texttt{level}(1) = \{f \mid \exists \sigma \in \mathbb{S}[\![\tau]\!] : \sigma.\texttt{file} = f\}$
- $\texttt{level}(2) = \{m \mid \exists \sigma \in \mathbb{S}[\![\tau]\!] : \sigma.\texttt{method} = m\}$
- $\texttt{level}(3) = \{l \mid \exists \sigma \in \mathbb{S}[\![\tau]\!] : \sigma.\texttt{loc} = l\}$
- $\texttt{level}(4) = \{t[\![l]\!] \mid \exists \sigma \in \mathbb{S}[\![\tau]\!] : \sigma.\texttt{loc} = l \wedge \sigma.\texttt{tmplt} = t\}$
- $\texttt{level}(5) = \{t[\![l]\!][\mathrm{i}] \mid \exists \sigma \in \mathbb{S}[\![\tau]\!] : \sigma.\texttt{loc} = l \wedge \sigma.\texttt{tmplt} = t \wedge \sigma.\texttt{idx} = i\}$
- $\texttt{level}(6) = \{\sigma \mid \sigma \in \mathbb{S}[\![\tau]\!]\}$

At level 4, $t[\![l]\!]$ represents a template t applied at location l. At level 5, $t[\![l]\!][i]$ represents an index of σ among the $t[\![l]\!]$-using patches. Connecting an *edge* between nodes is straightforward. Given a patch configuration $\sigma = \langle f, m, l, t, i\rangle \in \mathbb{S}[\![\tau]\!]$ and the root node r, we construct a path $r \to f \to m \to l \to t[\![l]\!] \to t[\![l]\!][i] \to \sigma$ in $\mathcal{T}[\![\tau]\!]$. We denote this path with $\pi(\sigma)$.

Notation. To represent the *subtree* rooted from a component c of a patch configuration σ, we write $\mathcal{T}[\![\tau]\!](c)$. For example, in Figure 1, the left-most subtree is denoted with $\mathcal{T}[\![\tau]\!](m_1)$.

Notation. We will omit τ in $\mathbb{S}[\![\tau]\!]$ and $\mathcal{T}[\![\tau]\!]$ when we are not concerned about which APR tool τ is used.

3.1.2 List Structure. Given a subtree $\mathcal{T}(c)$ and the patches $\sigma \in \mathcal{T}(c)$, we maintain a hierarchy of groups. See Figure 1 where subtree $\mathcal{T}(m_1)$ is transformed into the hierarchy of three groups. Each group consists of the patches having the same suspicious score (denoted with `s-score`) obtained through fault localization. In most APR, `s-score` is computed at the location level and we assume that $\texttt{s-score}(\sigma) = \texttt{s-score}(l)$; that is, the `s-score` computed for l represents the `s-score` of σ. Now, we define a patch-space list.

Definition 4 (Patch-Space List $\mathbb{L}$)**.** Given a patch-space subtree $\mathcal{T}(c)$ rooted from component c, it holds that $\mathcal{T}(c) = \biguplus \mathbb{L}_i(c)$ where $\uplus$ is the disjoint-union operator and $\mathbb{L}_i(c)$ satisfies the following:

(1) $\forall i : \{\sigma \mid \sigma \in \mathbb{L}_i(c)\} \subseteq \{\sigma \mid \sigma \in \mathcal{T}(c)\}$
(2) $\exists n : \bigcup_{1 \le i \le n} \{\sigma \mid \sigma \in \mathbb{L}_i(c)\} = \{\sigma \mid \sigma \in \mathcal{T}(c)\}$
(3) $\forall i, j : i \neq j \Rightarrow \mathbb{L}_i(c) \cap \mathbb{L}_j(c) = \emptyset$
(4) $\forall \sigma_1, \sigma_2 \in \mathbb{L}(c) : \texttt{s-score}(\sigma_1) = \texttt{s-score}(\sigma_2)$.
(5) $\forall i, j : i \neq j \Rightarrow \texttt{s-score}(\mathbb{L}_i(c)) \neq \texttt{s-score}(\mathbb{L}_j(c))$ where $\texttt{s-score}(\mathbb{L}_i(c))$ represents the `s-score` of $\sigma \in \mathbb{L}_i(c)$.

In Figure 1, $\mathcal{T}(m_1) = \mathbb{L}_1(m_1) \uplus \mathbb{L}_2(m_1) \uplus \mathbb{L}_3(m_1)$ where $\mathbb{L}_i(m_1)$ consists of the patches $\sigma \in \mathcal{T}(m_1)$ having $\texttt{s-score}_i$. The reason we call $\mathbb{L}_i$ a list, not a group, will be clear in Definition 7. We will omit c in $\mathbb{L}(c)$ when the current component c is clear from the context.

Notice in Figure 1 that $\texttt{s-score}_1 > \texttt{s-score}_2 > \texttt{s-score}_3$. We order patch-space lists based on their `s-score`.

Definition 5 (Ordering between Patch-Space Lists)**.** Given two patch-space lists $\mathbb{L}_1$ and $\mathbb{L}_2$ belonging to the same patch-space subtree, we write $\mathbb{L}_1 \sqsupset \mathbb{L}_2$ iff $\texttt{s-score}(\mathbb{L}_1) > \texttt{s-score}(\mathbb{L}_2)$.

While scheduling a patch, we want to avoid selecting a patch that was already selected (i.e., covered) before. For example, in Figure 1, we do not want to select a patch in the $\texttt{s-score}_1$-group if all patches in that group were already covered. If there exists an uncovered patch in the $\texttt{s-score}_1$-group, CASINO skips the $\texttt{s-score}_1$-group and considers the $\texttt{s-score}_2$-group. In this case, we call the $\texttt{s-score}_2$-group *the MSU (most suspicious uncovered)* list.

Definition 6 (The Most Suspicious Uncovered (MSU) List)**.** The MSU list denoted with $\mathbb{L}_{MSU}$ satisfies the following constraints.

(1) $\forall i : \mathbb{L}_i \sqsupset \mathbb{L}_{MSU} \Rightarrow \forall \sigma \in \mathbb{L}_i : \sigma.\texttt{cvr} = \text{true}$
(2) $\exists \sigma \in \mathbb{L}_{MSU} : \sigma.\texttt{cvr} = \text{false}$

As mentioned, all patches in a patch-space list have the same `s-score`. In most APR tools, patches are scheduled in a fixed specific order, even among those having the same `s-score`. Considering this, we order the patches in the same patch-space list.

Definition 7 (Ordering between Patches)**.** Consider the patch space $\mathbb{S}[\![\tau]\!]$ of an APR tool τ. Given two patches $\sigma_1, \sigma_2 \in \mathbb{S}[\![\tau]\!]$, we write $\sigma_1 \sqsupset \sigma_2$ when the original scheduling algorithm of τ validates σ_1 before σ_2. We preserve this original ordering between patches in each patch-space list $\mathbb{L}$. That is, $\forall i, j : i > j \Rightarrow \mathbb{L}[\![i]\!] \sqsupset \mathbb{L}[\![j]\!]$ where $\mathbb{L}[\![i]\!]$ and $\mathbb{L}[\![j]\!]$ denote the i-th and j-th patch of $\mathbb{L}$, respectively.

Notation. For simplicity, we will often use the words *tree* and *list* to refer to the patch-space tree and patch-space list.

3.2 Structured Navigation: Overview

To navigate the patch space, we use both the patch-space tree and patch-space list, taking the following four steps.

Step 1) Vertical Navigation. We first start with walking down the tree from the root. At each node of the tree, an edge to traverse is randomly chosen. We call this process *vertical navigation*. Vertical navigation can stop at any level l of the tree where $l \ge 4$ (i.e., traversal cannot occur below the template level). Once vertical navigation stops at node n, the search space is narrowed down to $\mathcal{T}(n)$.

Step 2) Extracting the MSU List. Subsequently, we extract the MSU list of $\mathcal{T}(n)$. This step further narrows down the scope of the search space into the extracted MSU list, $\mathbb{L}_{MSU}$.

Step 3) Horizontal Navigation. We randomly select the next patch among $\mathbb{L}_{MSU}$. We call this process *horizontal navigation*.

Step 4) Updating the Patch Space Once a chosen patch σ is validated, we set $\sigma.\texttt{cvr}$ to true to remove σ from the patch space.

3.3 Intermezzo: Multi-Armed Bandit

To efficiently explore the patch space, CASINO uses two MAB (multi-armed-bandit) algorithms described in this section.

3.3.1 ϵ-Greedy Algorithm. The ϵ-greedy algorithm [51] chooses an arm at time t in the following way:

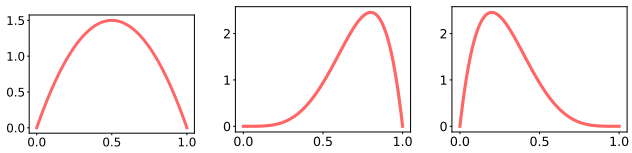
- choose an arm uniformly at random with probability ϵ
- choose the arm that yielded the highest profit in the past (i.e., until time $t-1$) with probability $1-\epsilon$.

3.3.2 Thompson Sampling Algorithm. CASINO uses the Thompson sampling algorithm [48, 53] for the Bernoulli bandit problem, where the reward for an arm is either 1 (success) or 0 (failure). In the Bernoulli bandit problem, each arm k produces a reward of one with probability θ_k and a reward of zero with probability $1-\theta_k$. While θ_k is unknown to the agent, the agent can learn about it through experimentation using a Bayesian approach. Suppose at time t, a prior distribution of $f(\theta_k)$ is defined as a two-parameter Beta distribution $Beta(\alpha_k, \beta_k)$ where $\alpha_k, \beta_k > 0$.

The useful property of the Beta distribution is that the posterior distribution after observing a reward (either 1 or 0) is again a Beta distribution where α_k and β_k are updated as follows:

$$(\alpha_k, \beta_k) := \begin{cases} (\alpha_k + 1, \beta_k) & \text{if success is observed} \\ (\alpha_k, \beta_k + 1) & \text{if failure is observed} \end{cases}$$

At time t, having observed n successes and m failures from arm k, the posterior distribution of k is $Beta(1+n, 1+m)$. See Figure 2 shows examples of Beta distributions with different parameters.



(a) $\alpha = 2, \beta = 2$ **(Balanced)** **(b)** $\alpha = 5, \beta = 2$ **(Left-skewed)** **(c)** $\alpha = 2, \beta = 5$ **(Right-skewed)**

Figure 2: Examples of Beta distributions

Thompson sampling for the Bernoulli bandit problem repeats the following three steps at each time step t.

(1) **Sampling:** For each arm k, success probability estimate θ_k is randomly drawn from $Beta(\alpha_k, \beta_k)$.

(2) **Selection:** The algorithm selects the arm for which the largest success probability estimate is sampled. After playing the chosen arm x_t, its reward r_t is observed.

(3) **Update:** Based on the observation r_t, the Beta distribution is updated accordingly.

3.4 Patch Scheduling via Multi-Armed Bandit

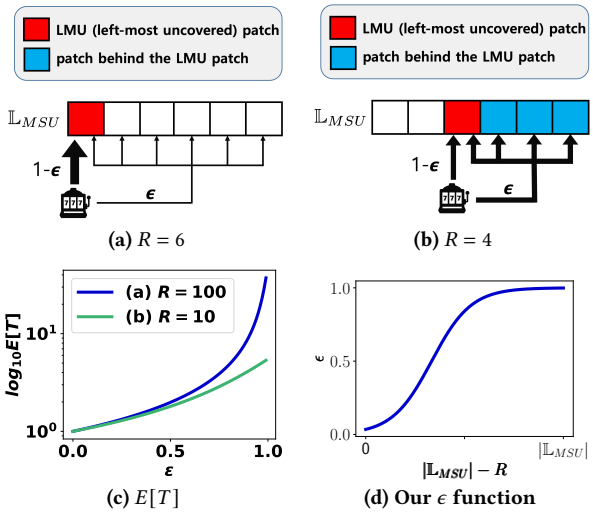
3.4.1 Labeled Patch-Space Tree. To explore the patch-space tree $\mathcal{T}$ stochastically, we extend $\mathcal{T}$ by labeling edges e_k with Beta distributions $Beta(\alpha_k, \beta_k)$. Hereafter, the patch-space tree and $\mathcal{T}$ refer to the labeled patch-space tree.

3.4.2 Vertical Navigation via Thompson Sampling. At each level of $\mathcal{T}$, we choose an edge randomly using Thompson sampling (§ 3.3.2). The edge labeled with a Beta distribution skewed more to the left is more likely to be selected. Conversely, when we observe evidence to believe that a certain edge should be more prioritized than before, we further skew the Beta distribution of that edge to the left by increasing the α value of the distribution. More details will be described in § 3.4.6.

3.4.3 Initial Edge label $\perp$. Before starting an APR campaign, we have no runtime information to believe that one edge e_1 looks more profitable than another edge e_2 existing at the same level of the tree. Thus, we initialize all edge labels of $\mathcal{T}$ with a special value, $\perp$. When performing Thompson sampling, the edges e labeled with $\perp$ are ignored and not selected. We say that those edges e are *disabled*. We will describe in § 3.4.5 when an edge is enabled.

3.4.4 Horizontal Navigation via ϵ-Greedy Algorithm. Since all edges are disabled in the beginning, vertical navigation over $\mathcal{T}[\![\tau]\!]$ immediately stops at the root node r of $\mathcal{T}[\![\tau]\!]$ and returns subtree $\mathcal{T}[\![\tau]\!](r)$. As described in § 3.2, horizontal navigation is performed over the MSU list of $\mathcal{T}[\![\tau]\!](r)$. Since subtree $\mathcal{T}[\![\tau]\!](r)$ refers to $\mathcal{T}$ and no patch is covered yet, the MSU list of $\mathcal{T}[\![\tau]\!](r)$ is the list consisting of the patches having the highest `s-score`.

Given the MSU list $\mathbb{L}_{MSU}$, which item in it should we select as the next patch to validate? On the one hand, it seems we should use random selection since all patches in the list have the same `s-score`. On the other hand, it also seems reasonable to select the left-most uncovered item of $\mathbb{L}_{MSU}$ as done in most APR tools. Many APR tools distinguish the ordering between the patches having the same `s-score` by using additional information. For example, learning-based tools use the patch likelihood returned from a trained model [62, 72].



(c) $E[T]$ **(d) Our ϵ function**

Figure 3: Horizontal Navigation via ϵ-Greedy Algorithm. In (a) and (b), a thicker arrow means a higher probability. $E[T]$ R are defined in § 3.4.4 with signpost "$E[T]$ and R".

Reconciling both factors, we use an ϵ-greedy algorithm to perform horizontal navigation. Consider Figure 3(a). Our algorithm allows both (1) random selection and (2) the selection of the *LMU (left-most uncovered) patch*, σ_{LMU}. The former is selected with the probability of ϵ and the latter with $1 - \epsilon$. It can be viewed that with the probability of ϵ, we prioritize "exploration", whereas with the probability of $1 - \epsilon$, we "exploit" the pre-defined patch ordering imposed by the APR tool. After discovering an "interesting" patch (see Definition 8) as a result of exploration, our scheduling algorithm prioritizes its similar patches, just as a fuzzer prioritizes inputs similar to interesting seeds.

E[T] and R. When performing horizontal navigation using an ϵ-greedy algorithm, the balance between exploitation and exploration is controlled using the value of ϵ. To define ϵ, we consider the following. Let T be the number of trials until σ_{LMU} is selected. Then, the expected value $E[T]$ is $\sum_{k=1}^{R} \epsilon^{k-1} \times \frac{R-k+1}{R} \times (1 - \epsilon + \frac{\epsilon}{R-k+1})$ where $R = |\{\sigma \mid idx(\sigma) \geq idx(\sigma_{LMU})\}|$ and idx denotes a function that returns the index of the given element of $\mathbb{L}_{MSU}$.[2] Notice that R represents the size of the sublist of $\mathbb{L}_{MSU}$ ranging from σ_{LMU} to the last item of $\mathbb{L}_{MSU}$. The plot of $E[T]$ is shown in Figure 3(c). $E[T]$ increases exponentially as ϵ increases. Also, the plot grows more rapidly when the value of R is larger.

The value of ϵ should not be too large or too small. Suppose σ_{LMU} is valid (i.e., passes all tests), and thus the original scheduling algorithm detects a valid patch σ_{LMU} in the next scheduling iteration. In comparison, our algorithm would detect the same σ_{LMU} in $E[T]$ iterations on average, incurring the additional average cost of $E[T] - 1$. Thus, if $E[T]$ is too large, the cost is likely to surpass the

[2] § A.1 of the supplementary appendix describes how we derive this formula of $E[T]$.

gain. Conversely, if $E[T]$ is too small, exploration would not occur frequently enough to obtain gains.

Unlike the conventional ϵ-greedy algorithm, we do not use a single constant for ϵ. This is because as R decreases (i.e., the position of σ_{LMU} moves to the right in $\mathbb{L}_{MSU}$; see Figure 3(b)), $E[T]$, the expected cost of the random search, also decreases (see Figure 3(c)), and thus we can afford a larger ϵ value. We define ϵ as the sigmoid function (see Figure 3(d)) defined in Equation (1). Notice that as R decreases, i.e., $|\mathbb{L}_{MSU}| - R$ increases, ϵ increases as desired.

$$\epsilon(x) = \frac{1}{1 + e^{\frac{c_1}{|\mathbb{L}_{MSU}|} \cdot (c_2 \cdot |\mathbb{L}_{MSU}| - x)}} \quad (1)$$

In Equation (1), constants c_1 and c_2 are set to 10 and 1/3, respectively, to form a balanced S shape.[3]

3.4.5 Enabling Edges. As mentioned in § 3.4.3, all edges of $\mathcal{T}$ are disabled initially. We enable an edge when we detect an interesting patch defined as follows:

Definition 8 (Interesting Patch)**.** A patch σ is considered interesting when the program p patched with σ passes one of the negative tests (i.e., the test p previously failed).

Consider Figure 1 as an example. Given a detected interesting patch $\sigma_\star$, we identify $\pi(\sigma_\star) = r \rightarrow f_\star \rightarrow m_\star \rightarrow l_\star \rightarrow t_\star[\![l_\star]\!] \rightarrow t_\star[\![l_\star]\!][i_\star] \rightarrow \sigma_\star$.[4] Then, we assign a Beta distribution $Beta(3, 2)$ to the red-colored edges of $\pi(\sigma_\star)$. We assign $Beta(3, 2)$ not $Beta(2, 2)$ to reflect the fact that an interesting patch is found in path $\pi(\sigma_\star)$. Recall that we increase the α value of the Best distribution to give a higher priority than before. The edges below the template level remain disabled to enforce the horizontal navigation below the template level.

Just as a fuzzer prioritizes inputs similar to interesting seeds, our patch-scheduling algorithm prioritizes patches similar to detected interesting patches $\sigma_\star$. To allow selecting edges deviating from $\pi(\sigma_\star)$, we define the neighboring edge as follows.

Definition 9 (Neighboring Edge and Node of $\sigma_\star$)**.** Given an interesting patch $\sigma_\star$, its neighboring edge e is defined as follows:

$$e \in \{e \mid n \in \pi(\sigma_\star) \wedge e \text{ is an outgoing edge of } n \wedge e \notin \pi(\sigma_\star)\}$$

where n is a node in path $\pi(\sigma_\star)$. When a node is visited via a neighboring edge, we call that node a neighboring node.

In Figure 1, blue-colored edges are neighboring edges of $\pi(\sigma_\star)$. We label the neighboring edges of $\pi(\sigma_\star)$ with $Beta(2, 2)$ shown in Figure 2(a). As a result, in our running example, the vertical navigation can visit neighboring nodes like m_1 and l_2. Once an edge is enabled, we do not disable it afterward.

3.4.6 Updating Edge Labels. As explained § 3.4.5, a disabled edge is enabled when an interesting patch $\sigma_\star$ is detected. What if some edges in $\pi(\sigma_\star)$ are already enabled? Given an edge $e \in \pi(\sigma_\star)$ labeled with $Beta(\alpha, \beta)$, we update the label of e to $Beta(\alpha + n, \beta)$ where $n > 0$. The resultant Beta distribution is skewed more to the left than before, increasing the chance to be selected when performing vertical navigation. We increase the α value exponentially (i.e., n is 2, 4, 8, 16, . . .) as the edge is covered by a larger number of interesting patches. However, we do not change the β value when the scheduled patch is not interesting. Most patches are not interesting and thus increasing the β value makes all Best distributions of the edges similar to each other. Our approach is similar to that used in fuzzing. Most fuzzers keep and mutate interesting inputs while discarding uninteresting ones.

[3] § A.2 in the supplementary appendix shows the shapes of the ϵ function when different values are used.

[4] In Figure 1, we simply write $t_\star$ and $i_\star$ to represent $t_\star[\![l_\star]\!]$ and $t_\star[\![l_\star]\!][i_\star]$, respectively.

Algorithm 1 CASINO Patch Scheduling Algorithm

Input: P: Program to be repaired
Input: Initial patch-space tree $\mathcal{T}$
Input: O_{fix}: Fix oracle (e.g., a test-suite)
Output: $\mathbb{D}_{\text{patches}}$

1: $\mathbb{D}_{\text{patches}} \leftarrow \emptyset$ // $\mathbb{D}_{\text{patches}}$: Set of detected patches
2: **while** $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ CONTINUE() **do**
3: // Vertical navigation via Thomson sampling (see § 3.4.2)
4: $n :=$ ROOT($\mathcal{T}$) // Start patch-tree traversal from the root node
5: **while** n has an enabled outgoing edge **do**
6: **for** each enabled edge e of n **do**
7: $Beta(\alpha_k, \beta_k) \leftarrow$ LABEL(e)
8: Draw $\hat{\theta}_k$ according to $Beta(\alpha_k, \beta_k)$
9: **end for**
10: $e := \arg\max_k \hat{\theta}_k$ // Select the k-th edge having the maximum $\hat{\theta}_k$
11: $n :=$ TRAVERSE(n, e) // Traverse edge $n \xrightarrow{e} n'$ and $n := n'$
12: **end while**
13: // Horizontal navigation via ϵ-greedy algorithm (see § 3.4.4)
14: $\mathbb{L}_{MSU} \leftarrow$ EXTRACTMSULIST($\mathcal{T}(n)$)
15: Toss a coin with success probability ϵ // see Eq (1)
16: **if** success **then**
17: $\sigma_{next} \leftarrow$ CHOOSEATRANDOM($\mathbb{L}_{MSU}, idx(\sigma_{LMU})$)
18: **else**
19: $\sigma_{next} \leftarrow \sigma_{LMU}$
20: **end if**
21: patch, execinfos $\leftarrow$ PATCHEVAL($P, \sigma_{next}, O_{\text{fix}}$)
22: **if** patch $\neq \perp$ **then** // If a valid patch is found
23: $\mathbb{D}_{\text{patches}} \leftarrow \mathbb{D}_{\text{patches}} \cup \{\text{patch}\}$
24: **end if**
25: // Update edge labels (see § 3.4.5 and 3.4.6)
26: UPDATE(execinfo, $\mathcal{T}, \mathbb{L}_{MSU}, \sigma_{next}$)
27: **end while**

3.4.7 Putting All Together. Algorithm 1 puts together the patch-scheduling components described in § 3.4. The outermost loop of the algorithm (lines 2–27) collects valid patches. The program location to modify and the mutation operator to use are scheduled using the combination of vertical navigation (lines 3–12) and horizontal navigation (lines 13–20). Once the scheduled patch σ_{next} is validated, we update $\mathcal{T}$ and σ_{next} (lines 21–26).

3.5 On-The-Fly Construction of Patch Space

The CASINO algorithm does not require pre-generating patches. Instead, it constructs a patch-space tree on the fly. Consider an example shown in Figure 4 where we describe the first four steps of using Algorithm 1.

Step 1) Given a fault localization result (which is obtained before patch scheduling begins), we prepare an initial patch-space tree. Note that the initial patch space tree does not contain any patches yet, since it is constructed only with the information available

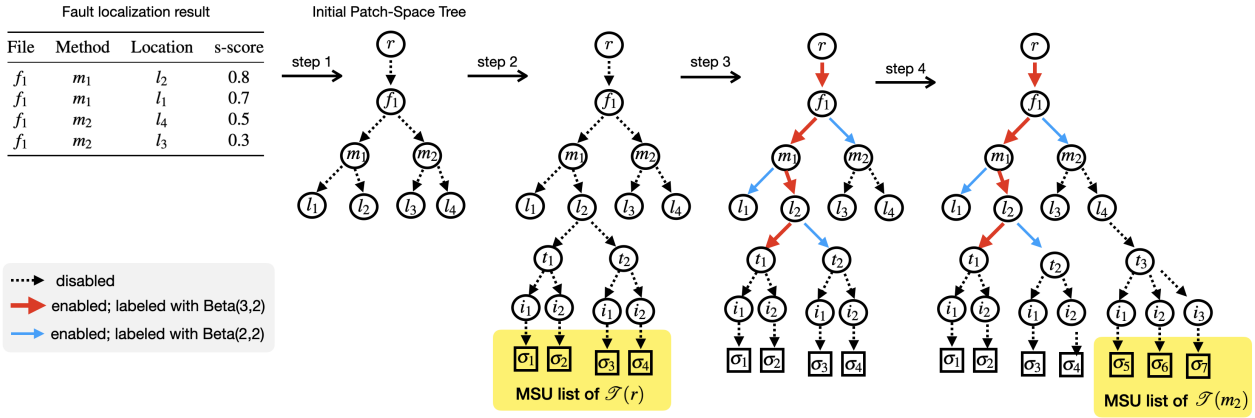


Figure 4: On-The-Fly Construction of Patch Space (§ 3.5)

in the fault localization result (i.e., files, methods, and locations). Also, note that all edges are disabled, indicating that no interesting patch has been found yet. We pass this initial patch-space tree to Algorithm 1.

Step 2) Since all edges of the tree are disabled, the vertical navigation (line 4–12) immediately finishes without entering the loop in lines 5–12. Subsequently, the horizontal navigation is performed with the MSU list of $\mathcal{T}(r)$ where r is the root node of $\mathcal{T}$. Recall that $\mathcal{T}(n)$ represents the sub-tree rooted at node n. In the running example, the MSU list of $\mathcal{T}(r)$ consists of σ_1, σ_2, σ_3, and σ_4 — the patches the APR tool generates for location l_2 whose suspiciousness score (i.e., `s-score`) is highest (i.e., 0.8), assuming that two templates t_1 and t_2 are applicable at l_2. Note that the patches for the other locations are not generated.

Step 3) Suppose at step 2, patch σ_2 is randomly chosen and, subsequently, it is shown to be an interesting patch. Then, the edges leading to σ_2 and their neighboring edges are enabled as described in § 3.4.5.

Step 4) Suppose path $r \rightarrow f_1 \rightarrow m_2$ is taken while performing the vertical navigation. Since m_2 has no outgoing enabled edge, the horizontal navigation is performed with the MSU list of $\mathcal{T}(m_2)$, i.e., σ_5, σ_6, and σ_7 whose suspiciousness score is highest (i.e., 0.5) among the locations of method m_2.

As shown in this example, Casino incrementally builds the patch space on the fly, similar to the conventional APR tools.

4 EXPERIMENTAL DESIGN AND SETUP

4.1 Research Questions

To evaluate our approach Casino, we ask the following four research questions:

- **RQ1 (Search Efficiency)**: How efficiently does Casino find valid patches? Refer to § 4.2 for the definition of valid patches.
- **RQ2 (Recall)**: How many versions are successfully repaired? Refer to § 4.3 for the definition of a successful repair.
- **RQ3 (Ablation Study)**: Casino combines the vertical and horizontal navigations. How does each navigation method contribute to the overall performance?
- **RQ4 (Generalizability)**: What is the generalizability of Casino?

4.2 Classification of Patches

To assess our research questions, we use several classifications of patches.

Valid Patch. We call a patch valid when that patch passes all tests in the given test suite. A valid patch is also called a *plausible patch* in the literature of APR.

Acceptable Patch. To assess RQ2, we need to determine the correctness of a generated patch. In the literature of APR, patch correctness has been determined either manually or using an additional test suite typically generated using an automated test generation tool. The former is prone to subjectivity and difficult to be applied when there are many patches to review. Meanwhile, the latter has little issue with the applicability but it may fail to identify incorrect patches. In our experiments, we obtain thousands (2,775) of valid patches[5] and resort to an automatic approach, similar to [50, 60]. To mitigate the risk of failing to identify incorrect patches, we use the state-of-the-art differential testing tool, DiffTGen [64], designed specifically to detect incorrect patches.[6] If no semantic difference is found between an obtained valid patch and its fixed version, we classify that patch as an **acceptable patch**.

4.3 Evaluation Methods

For RQ 1 and RQ 2, we compare Casino with the original APR tools (see § 4.4) over which Casino is applied. We also compare Casino with (1) SeAPR [6], the latest state-of-the-art patch-scheduling algorithm for APR and (2) GenProg^{SL}, an *APR*^{SL} version of the

[5]This is the result from Defects4J v.1.2. For the additional bugs in Defects4J v.2.0, we obtain 5,645 patches.
[6]The specific features of DiffTGen to detect incorrect patches are provided in § A.3 of the supplementary appendix. The DiffTGen configuration we used is described in § A.4.

GenProg family algorithm. More details about these comparison algorithms are provided in § 4.5.

For each buggy version, we run Casino and the other scheduling algorithms in separate sessions for 5 hours following recent work [27, 35, 49, 62, 72]. We run the two stochastic algorithms, Casino and GenProg^{SL}, 50 times, while SeAPR using a deterministic scheduling algorithm is run once.

RQ1 (Search Efficiency). We measure search efficiency by counting the number of valid patches over time.[7] For two stochastic algorithms, we compute the mean value of the results from 50 runs. We say that an algorithm A_1 outperforms A_2 when (1) A_1 finds valid patches at a faster rate than A_2 and (2) A_1 detects at least as many valid patches as found by A_2 within a time limit (i.e., 5 hours).

RQ2 (Recall). To measure recall, we count the number of buggy versions that are successfully repaired. We say that an APR tool τ successfully repairs a buggy version when the patches generated by τ include an acceptable patch. In this work, we consider the APR approach, where the generated patches are ranked in the final step using a ranking method (see § 2.2), and recall is computed for the top-N patches.

To perform ranking, we use ODS [67], the state-of-the-art patch classification system. ODS classifies a patch as either correct or incorrect using a machine-learned (ML) model trained with historical bug patches. We rank a patch based on its correctness likelihood returned by the ML model of ODS. Please note that in this work, we do not propose a new patch ranking/classification method, which itself is another active research topic of APR [8, 17, 54, 57, 65, 67]. Our goal here is to assess RQ2 with an existing patch ranking technique.

RQ3 (Ablation Study). We prepare two variations of Casino, i.e., Casino without vertical navigation (lines 5–12 of Algorithm 1 are omitted) and Casino without horizontal navigation (the ϵ function is set to always return 0). Then we compare their performances with Casino that uses both vertical and horizontal navigations.

RQ4 (Generalizability). As detailed in § 4.4, we evaluate Casino with a fresh benchmark not used in RQ1–3.

4.4 Evaluation Dataset and Subject APR Tools

To evaluate RQ1–3, we use Defects4J v1.2 [22], considering its widespread use for APR studies [23, 29–32]. Defects4J v1.2 consists of 395 bugs from 6 open-source software projects. To evaluate RQ4 about generalizability, we use the 440 additional bugs added into Defects4J v2.0, an extension of Defects4J v1.2.

As for APR tools, we consider the template-based approach and the learning-based approach used in the currently best-performing APR tools such as TBar [31] and AlphaRepair [62]. Given that most APR tools including those best-performing ones are APR^{SL} (i.e., they modify only a single location of the program), we in this study focus on APR^{SL}. Based on the aforementioned criteria, our subject APR tools include (i) the four template-based APR tools, i.e., TBar [31], Avatar [30], FixMiner [23], and kPar [29] covering all source-code-level template-based approaches used in recent studies on APR [4–6, 32] and (ii) two state-of-the-art learning-based APR tools, Recoder [72] and AlphaRepair [63]. We use all levels of the patch-space tree (Definition 3) across all four template-based tools and for the two learning-based APR tools, we omit the template level since those tools do not explicitly use templates.

[7] We also report the performance over *scheduling iterations* in § A.6 of the supplementary appendix.

4.5 Compared Algorithms

In addition to comparing the performance of Casino with the original APR tools, we consider two existing scheduling algorithms in our comparison: (1) GenProg^{SL}, a variant of GenProg to conduct APR^{SL} (§ 4.5.1) and (2) SeAPR [6], the latest patch scheduling algorithm (§ 4.5.2).

4.5.1 GenProg^{SL}. GenProg^{SL} repeats to mutate the original program by applying a random mutation operator (chosen at uniformly random) to a random location (chosen at random proportional to the suspicious score of the location). The concept of generational evolution is not used because in APR^{SL}, only a single-step modification is allowed.

4.5.2 SeAPR. SeAPR assigns a priority score (`p-score`) to each patch in a similar way to how spectrum-based fault localization (SBFL) assigns a `s-score` to each program element. Intuitively, SeAPR assigns a higher `p-score` to a patch that is more similar to an interesting patch, or a high-quality patch if we use the terminology of SeAPR. Conversely, a lower priority score is assigned to a patch that is more similar to a low-quality patch – i.e., a patch that cannot make any originally failing test pass. Two patches are considered similar to each other when they modify the same program elements such as methods. SeAPR computes `p-score` using an SBFL formula; by default, SeAPR uses the Ochiai formula [1].

4.6 Experimental Setup through Simulation

In our experiments, we run 6 APR tools with 4 scheduling algorithms described earlier. We run two stochastic algorithms, GenProg^{SL} and Casino, 50 times considering their randomness. To evaluate RQ3 (ablation study), we run two variations of Casino 50 times. For each version of Defects4J v1.2 containing 395 bugs, we use a 5-hour timeout. Lastly, to evaluate RQ4 (generalizability), we run the original scheduling algorithms of the six APR tools and Casino 50 times again for 440 new buggy versions in Defects4J v2.0. All these numbers add up to 3,073,020 hours (≈ 350 years).

SimAPR. To conduct experiments in a shorter time frame, we implement a replay-based patch-scheduling simulation system named SimAPR. Figure 5 shows how SimAPR interacts with the other modules of APR to search for valid patches. SimAPR takes as input (1) a buggy program P_b, (2) a ranked list of suspicious program locations of P_b, and (3) a subject patch-scheduling algorithm. To obtain a ranked list of suspicious program locations, we use the latest release of GZoltar [7] v1.7.3 with the Ochiai [1] formula. We do not use the FL (fault localization) module of the original APR tool to use the same FL result across all six subjects.

Given an input, SimAPR runs the patch-scheduling algorithm to choose the program location l to modify. Then, SimAPR retrieves the list of patches for l generated by the subject APR tool. To reduce patch generation time during simulation, we pre-generate the patches for all suspicious program locations in the offline phase

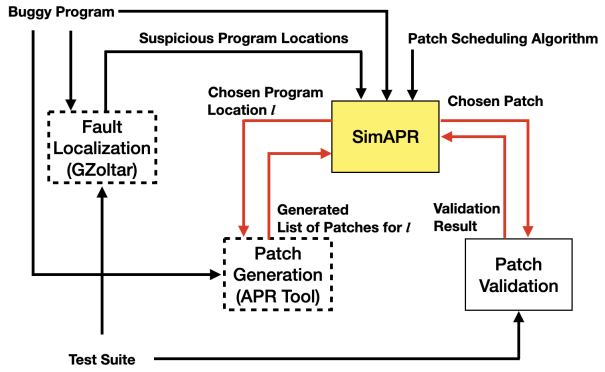


Figure 5: Interaction of SimAPR with the other modules. The dashed boxes represent offline modules (see § 4.6).

before running SimAPR. It should be noted that the patch generation modules of all six APR tools used in our experiments are deterministic. Therefore, given a program location l, each APR tool consistently generates the same list of patches for l. Once the list of patches is obtained, SimAPR chooses a patch to validate from the list while running the patch-scheduling algorithm. The selected patch is then validated using the test suite, similar to the original APR tools. To save validation time, we cache the validation result of each patch and reuse it in subsequent runs.[8] To store the cached validation results, we extend the definition of the patch configuration σ defined in Definition 1 as follows:

Definition 10 (Extended Patch Configuration)**.** Consider the patch configurations σ generated by an APR tool τ. An extended patch configuration σ_e extends σ (see Definition 1) with the following three additional attributes:

- `valid`: true if σ is a valid patch; false otherwise.
- $\texttt{time}_{gen}$: time taken for τ to generate the patch referred to by σ.
- $\texttt{time}_{val}$: time taken to validate σ with a given test suite. The validation process stops at the first failing test, as done in most APR tools, including our six subject tools.

Simulation of APR via SimAPR. We evaluate the performance of the patch-scheduling algorithms using SimAPR, inspired by FuzzSim [61], which compares the performance of various fuzzing algorithms via simulation. As mentioned, we perform a simulation in two phases, i.e., an offline phase followed by an online phase. In the offline phase, we conduct fault localization and generate patches for all suspicious program locations using the subject APR tools. We represent each patch as an extended patch configuration σ_e. When generating a patch, we measure the time and assign the obtained value to $\sigma_e.\texttt{time}_{gen}$. Meanwhile, the `valid` and $\texttt{time}_{val}$ attributes are initialized with $\perp$.

In the follow-up online phase, we run a subject patch-scheduling algorithm using SimAPR. For each scheduled patch σ_e, SimAPR performs the following depending on the value of $\sigma_e.\texttt{valid}$.

- **When** $\sigma_e.\texttt{valid} = \perp$: SimAPR validates σ_e with the given test suite and assigns the validation result (either true or false) to $\sigma_e.\texttt{valid}$. When validating σ_e, SimAPR measures the validation time and assigns the obtained value to $\sigma_e.\texttt{time}_{val}$.
- **When** $\sigma_e.\texttt{valid} \neq \perp$: SimAPR returns $\sigma_e.\texttt{valid}$, $\sigma_e.\texttt{time}_{gen}$ and $\sigma_e.\texttt{time}_{val}$ to the scheduling algorithm.

While running a subject scheduling algorithm in our experiments, we accumulate $\sigma_e.\texttt{time}_{gen}$ and $\sigma_e.\texttt{time}_{val}$ for every scheduled σ_e and count the number of detected valid patches.

Scheduling Algorithms. SimAPR currently supports four scheduling algorithms used in our experiment: (1) sequential scheduling that iterates over a given list of patches, (2) GenProg^{SL}, (3) SeAPR, and (4) Casino. The sequential scheduling is used to simulate the original scheduling order of a subject APR tool. The SeAPR algorithm available in SimAPR uses the configuration shown to work best in its original paper [6].[9]

Open Science. We expect that SimAPR will be useful for other researchers interested in studying patch-scheduling algorithms. In support of open science, we release the source code of SimAPR. With SimAPR, a new patch-scheduling algorithm can be easily evaluated.

Experimental Environment. We conducted experiments using a machine with AMD EPYC 2.6GHz CPUs (1024GB RAM) and another one with Intel Xeon Gold 3GHz CPUs (128GB RAM). The former was used for all subject tools except for FixMiner which was experimented on the latter machine. Please note that all four compared scheduling algorithms were run on the same machine. We used Ubuntu 20.04.4 LTS and Java 1.7 across all experiments.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Search Efficiency

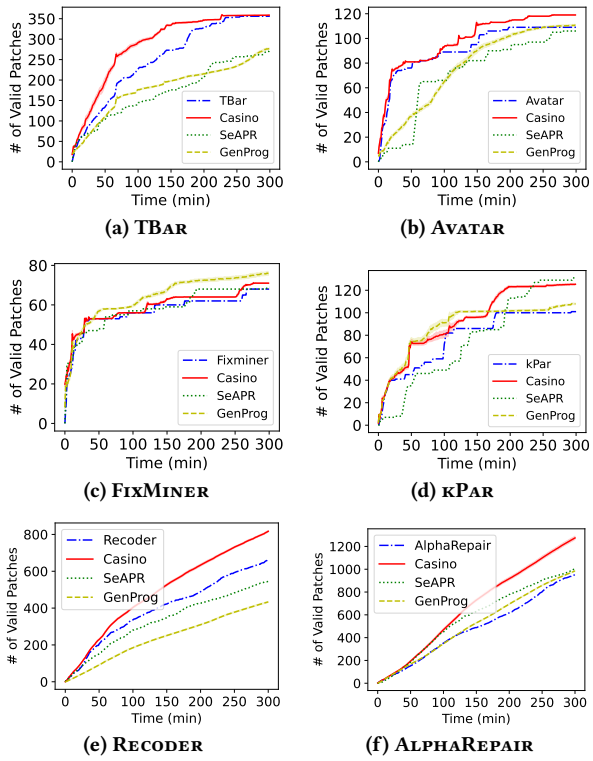
Figure 6 shows how many *valid* patches are detected over time during 300 minutes. For GenProg^{SL} and Casino, the mean values of the results from 50 runs are shown. We also draw shade around the lines for those two algorithms to show 95% confidence intervals (CI) for the cumulative number of patches at each time point. As indicated by the thin shades, only marginal variance is observed from 50 runs where at each run, we use a unique random seed. We observe the following from the figure.

(1) **Improved Performance.** Casino detects valid patches at faster rates than the original algorithms, as indicated by the red curves running above the blue ones, though the degree of improvement varies depending on an APR tool. Casino particularly performs well in TBar, Recoder and AlphaRepair. Notice that in these three tools, a larger number of valid patches are found than in the other tools, suggesting that Casino seems to work well when the patch space contains valid patches more abundantly. In terms of the total number of valid patches detected within a time limit, Casino discovers more (Avatar, kPar, Recoder and AlphaRepair) or similarly (TBar and FixMiner) as compared to the original algorithms.

(2) **Mixed Result of SeAPR and GenProg^{SL}.** Unlike Casino, the results of SeAPR and GenProg^{SL} are mixed. SeAPR outperforms the original algorithm in AlphaRepair and FixMiner. For kPar, SeAPR performs worse than the original algorithm in the

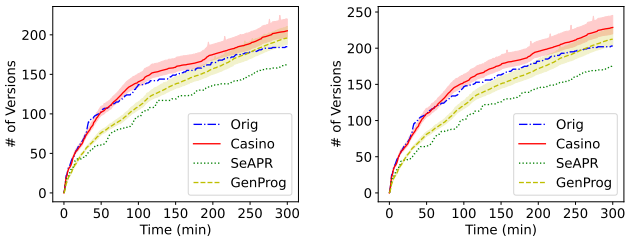
[8] Tests are assumed to be deterministic following the current convention of APR.

[9] More details are available in § A.5 of the supplementary appendix.



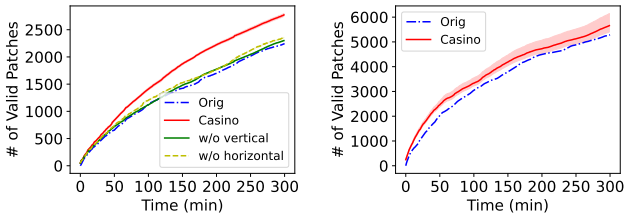
(a) TBar (b) Avatar (c) FixMiner (d) kPar (e) Recoder (f) AlphaRepair

Figure 6: Search efficiency in finding valid patches. In the legend, GenProg refers to GenProg^{SL}.



(a) Top-1 (b) Top-5

Figure 7: Recall at top-N when ranking is computed using ODS. The Y-axis shows the number of successfully repaired versions.



(a) Ablation Study (v.1.2) (b) Generalizability (v.2.0 − v.1.2)

Figure 8: Results for ablation study and generalizability

beginning but eventually finds more valid patches. In the three remaining tools (TBar, Avatar, and Recoder), SeAPR detects valid patches at a slower rate than the original algorithms. As for GenProg^{SL}, it outperforms the original algorithm in FixMiner, kPar, and AlphaRepair. However, the performance of GenProg^{SL} is worse than the original in TBar, Avatar, and Recoder.

(3) **Casino is a frequent winner.** Casino wins over SeAPR and GenProg^{SL} in the majority of the cases, including TBar, Avatar, Recoder and AlphaRepair. In FixMiner, GenProg^{SL} wins over Casino with a narrow margin. In kPar, there is no clear winner. While GenProg^{SL} performs best in the beginning, Casino and SeAPR eventually surpass GenProg^{SL}. SeAPR performs worst during most periods but wins closely at the almost last moment.

5.2 RQ2: Recall

Figure 7 shows how many *versions* are successfully repaired over time within the 300-minute period. At each plot of Figure 7, the coordinate (x, y) represents that within x minutes, y versions are successfully repaired when accumulating the results from the six APR tools. The shade around the red line (Casino) and the yellow line (GenProg^{SL}) represents the 95% confidence intervals (CI) for the cumulative number of valid patches at each time point.

The four plots of Figure 7 show the recall for top-N patches. As N increases, the number of successfully repaired versions increases (compare the Y-axes). The overall pattern is consistent across N. Only Casino outperforms the original algorithms — i.e., a repair succeeds more frequently and at a faster rate than in the original algorithms. In comparison, GenProg^{SL} succeeds in repairing bugs at a slower rate than the original algorithms, although eventually, it succeeds to repair more instances of buggy versions. Meanwhile, SeAPR performs worst in our experiments.

5.3 RQ3: Ablation Study

Figure 8(a) compares the performance of Casino with their two variances in lack of either vertical or horizontal navigation. In the figure, the coordinate (x, y) represents that within x minutes, the six subject APR tools generate y valid patches when those six tools run in parallel with separate instances of the same set of buggy versions. It is observed that to maximize performance, both vertical and horizontal navigations should be used.

5.4 RQ4: Generalizability

To assess the generalizability of Casino in terms of search efficiency, we reran SimAPR against the 440 new bugs in Defects4J v.2.0. Figure 8(b) shows how many valid patches are detected over time when the original algorithms and Casino are used. Compare Figure 8(b) with Figure 8(a). In both figures, Casino outperforms the original algorithms, implying the generalizability of Casino.

6 THREATS TO VALIDITY

Benchmarks and subject tools. As with all experimental results, caution should be taken when generalizing our results. To mitigate this threat, we evaluate the generalizability of Casino with Defects4J v.2.0.

Patch correctness. To validate a large number of patches, we took a best-effort approach by using the differential testing tool specialized for patch correctness validation. Nevertheless, DiffTGen may fail to detect incorrect patches. Nonetheless, the incorrect "acceptable" patches can be viewed as patches whose incorrectness cannot be detected easily.

7 RELATED AND FUTURE WORK

The search efficiency of APR is affected by many factors, including patch prioritization, patch space reduction, test case prioritization, and test-suite reduction.

Patch Prioritization. To prioritize patch candidates, existing APR approaches use various additional information such as code context [59], program contracts [44], code comments [66], Q&A sites [14], and existing patches [20, 24, 34]. These approaches typically build a statistical model and use it to guide the search for repair. Recent deep-learning-based APR [12, 21, 27, 28, 35, 63, 72] can be viewed as an extension of this line of work. Compared to them, Casino adjusts prioritization dynamically, similar to fuzzing. While GenProg [25] and its descendants [69, 70] also work similarly, those algorithms use genetic programming as a main vehicle, whereas Casino works in a more similar way to grammar-aware mutation-based fuzzing. Casino is a patch-scheduling algorithm for APR^{SL} and extending it to multi-line APR is a potential future work.

Meanwhile, semantics-based approaches [38, 39, 42, 68] do not directly search for patches. Instead, they first search for constraints that a patch must satisfy and then synthesizes a patch that satisfies the constraints. While most semantics-based approaches do not use dynamic information to guide the search, FAngelix [68] performs a stochastic search for constraints using MCMC (Markov Chain Monte Carlo) sampling [3]. Compared to this work, Casino performs a stochastic search for patches, not constraints.

Patch Space Reduction. To keep the patch space tractable, several patch space reduction techniques have been developed. These techniques exclude from the patch space harmful patch patterns (a.k.a., anti-patterns) [52], patch patterns involving intractably large search space [33], and patch candidates belonging to already covered test-equivalent classes [37]. Our patch-scheduling algorithm does not perform patch-space reduction and a combination of Casino and patch-space reduction technique is a promising direction to improve the search efficiency of APR.

Test Case Prioritization and Test-Suite Reduction. Running a test-driven APR tool typically involves the repetitive execution of tests. To reduce testing cost, test case prioritization [47] (which reveals incorrect patches as soon as possible) and regression test selection [13, 46] (which reduce the number of tests to run) have been developed and used in some APR tools [40, 45]. Casino and these techniques to manipulate a test suite are orthogonal and complementary to each other.

8 CONCLUSION

The researchers of APR have been faced with two seemingly conflicting goals. For the adoption of APR in the field, we need an APR technique that is both effective (fixing as many bugs as possible by enlarging the search space) and efficient (fixing them as soon as possible). In this paper, we have shown that it is possible to improve the search efficiency of APR for a given patch space.

The key ideas used in Casino— such as hierarchically structured patch space and stochastic patch-space navigation via multi-armed bandit — are inspired by recent advancements in fuzzing. In this paper, we have shown how fuzzing and APR are closely related to each other and used the obtained insight to design an efficient patch-space scheduling algorithm.

Our promising results encourage us to further seek a way to transfer fuzzing techniques to the APR side. Furthermore, we believe that some techniques of APR can be transferred to the fuzzing side for its improvement, completing a cycle of yin and yang.

ACKNOWLEDGEMENTS

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A5A1021944) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01001).

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. IEEE, Windsor, UK, 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[2] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous fuzzing for open source software. *Google Testing Blog* (2016).

[3] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. 2003. An introduction to MCMC for machine learning. *Machine learning* 50 (2003), 5–43.

[4] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: an extensive study on 16 program repair systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 907–918. https://doi.org/10.1145/3324884.3416566

[5] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2021. Evaluating and Improving Unified Debugging. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3125203

[6] Samuel Benton, Yuntong Xie, Lan Lu, Mengshi Zhang, Xia Li, and Lingming Zhang. 2022. Towards boosting patch execution on-the-fly. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2165–2176. https://doi.org/10.1145/3510003.3510117

[7] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. ACM Press, Essen, Germany, 378. https://doi.org/10.1145/2351676.2351752

[8] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding Automatically-Generated Patches Through Symbolic Invariant Differences. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, CA, USA, 411–414. https://doi.org/10.1109/ASE.2019.00046

[9] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. *URL: https://github. com/google/ossfuzz* (2016).

[10] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana, IL, 637–647. https://doi.org/10.1109/ASE.2017.8115674

[11] Liushan Chen, Yu Pei, and Carlo A. Furia. 2021. Contract-Based Program Repair Without The Contracts: An Extended Study. *IEEE Transactions on Software Engineering* 47, 12 (Dec. 2021), 2841–2857. https://doi.org/10.1109/TSE.2020.2970009

[12] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2019.2940179

[13] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. *Information and Software Technology* 52, 1 (2010), 14–30. Publisher: Elsevier.

[14] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Lincoln, NE, USA, 307–318. https://doi.org/10.1109/ASE.2015.81

[15] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 8–18. https://doi.org/10.1145/3293882.3330558

[16] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 19–30. https://doi.org/10.1145/3293882.3330559

[17] Ali Ghanbari and Andrian Marcus. 2022. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual South Korea, 654–665. https://doi.org/10.1145/3533767.3534368

[18] Patrice Godefroid. 2020. Fuzzing: hack, art, and science. *Commun. ACM* 63, 2 (Jan. 2020), 70–76. https://doi.org/10.1145/3363824

[19] Christian Holler, Kim Herzig, and Andreas Zeller. [n.d.]. Fuzzing with Code Fragments. ([n. d.]).

[20] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Amsterdam Netherlands, 298–309. https://doi.org/10.1145/3213846.3213871

[21] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, San Jose, CA, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[23] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Software Engineering* 25, 3 (May 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z arXiv:1810.01791 [cs].

[24] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, 213–224. https://doi.org/10.1109/SANER.2016.76

[25] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[26] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-Preserving Test Repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, China, 217–227. https://doi.org/10.1109/ICST.2019.00030

[27] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 602–614. https://doi.org/10.1145/3377811.3380345

[28] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: a novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 511–523. https://doi.org/10.1145/3510003.3510177

[29] Kui Liu, Anil Koyuncu, Tegawende F. Bissyande, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, China, 102–113. https://doi.org/10.1109/ICST.2019.00020

[30] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyande. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Hangzhou, China, 1–12. https://doi.org/10.1109/SANER.2019.8667970

[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, 31–42. https://doi.org/10.1145/3293882.3330577

[32] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 615–627. https://doi.org/10.1145/3377811.3380338

[33] 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*. ACM, Paderborn Germany, 727–739. https://doi.org/10.1145/3106237.3106253

[34] 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*. ACM, St. Petersburg FL USA, 298–312. https://doi.org/10.1145/2837614.2837617

[35] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 101–114. https://doi.org/10.1145/3395363.3397369

[36] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (Nov. 2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[37] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Transactions on Software Engineering and Methodology* 27, 4 (Nov. 2018), 1–37. https://doi.org/10.1145/3241980

[38] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 448–458. https://doi.org/10.1109/ICSE.2015.63

[39] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, Austin Texas, 691–701. https://doi.org/10.1145/2884781.2884807

[40] Ben Mehne, Hiroaki Yoshida, Mukul R. Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating Search-Based Program Repair. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Vasteras, 227–238. https://doi.org/10.1109/ICST.2018.00031

[41] Martin Monperrus. 2019. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan. 2019), 1–24. https://doi.org/10.1145/3105906

[42] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[43] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2228–2240. https://doi.org/10.1145/3510003.3510040

[44] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[45] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *2013 IEEE International Conference on Software Maintenance*. IEEE, Eindhoven, Netherlands, 180–189. https://doi.org/10.1109/ICSM.2013.29

[46] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551. Publisher: IEEE.

[47] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct. 2001), 929–948. https://doi.org/10.1109/32.962562

[48] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2020. A Tutorial on Thompson Sampling. http://arxiv.org/abs/1707.02038 arXiv:1707.02038 [cs].

[49] Seemanta Saha, Ripon k. Saha, and Mukul r. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[50] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. 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*. ACM, Bergamo Italy, 532–543. https://doi.org/10.1145/2786805.2786825

[51] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[52] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Seattle WA USA, 727–738. https://doi.org/10.1145/2950290.2950295

[53] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (1933), 285–294. Publisher: Oxford University Press.

[54] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 981–992. https://doi.org/10.1145/3324884.3416532

[55] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[56] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, Virtual. https://doi.org/10.14722/ndss.2021.24486

[57] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: how far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 968–980. https://doi.org/10.1145/3324884.3416590

[58] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, Vancouver, BC, Canada, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[59] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 1–11. https://doi.org/10.1145/3180155.3180233

[60] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens Greece, 354–366. https://doi.org/10.1145/3468264.3468600

[61] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. ACM Press, Berlin, Germany, 511–522. https://doi.org/10.1145/2508859.2516736

[62] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 959–971. https://doi.org/10.1145/3540250.3549101

[63] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair Via Zero-Shot Learning. http://arxiv.org/abs/2207.08281 arXiv:2207.08281 [cs].

[64] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Santa Barbara CA USA, 226–236. https://doi.org/10.1145/3092703.3092718

[65] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 789–799. https://doi.org/10.1145/3180155.3180182

[66] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, Buenos Aires, 416–426. https://doi.org/10.1109/ICSE.2017.45

[67] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2022. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Transactions on Software Engineering* 48, 8 (Aug. 2022), 2920–2938. https://doi.org/10.1109/TSE.2021.3071750

[68] Jooyong Yi and Elkhan Ismayilzada. 2022. Speeding up constraint-based program repair using a search-based technique. *Information and Software Technology* 146 (June 2022), 106865. https://doi.org/10.1016/j.infsof.2022.106865

[69] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (Oct. 2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

[70] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *ACM Transactions on Software Engineering and Methodology* 29, 1 (Jan. 2020), 1–53. https://doi.org/10.1145/3360004

[71] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. [n.d.]. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. ([n. d.]), 18.

[72] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens Greece, 341–353. https://doi.org/10.1145/3468264.3468544

[73] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s (Jan. 2022), 1–36. https://doi.org/10.1145/3512345