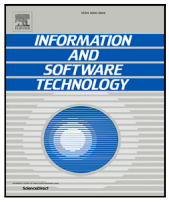
Information and Software Technology 146 (2022) 106865

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



# Speeding up constraint-based program repair using a search-based technique

Jooyong Yi *, Elkhan Ismayilzada

*UNIST, 50, UNIST-gil, 44919, Ulsan, Republic of Korea*

ARTICLE INFO

*Keywords:*
Automated program repair
Constraint-based program repair
Guided search
MCMC sampling

ABSTRACT

**Context:** Constraint-based program repair has been developed as one of the main techniques for automated program repair. Given a buggy program and a test suite, constraint-based program repair first extracts a repair constraint $\varphi$, and then synthesizes a patch satisfying $\varphi$. Since a patch is synthesized in a correct-by-construction manner (rather than compiling and testing each repair candidate source code), the constraint-based approach, in theory, requires less runtime overhead than the G&V approach. Nevertheless, the performance of existing constraint-based approaches is still suboptimal.

**Objective:** In this work, we propose a novel technique to expedite constraint-based program repair. We aim to boost runtime performance without sacrificing repairability and patch quality.

**Method:** The existing constraint-based program repair searches for a patch specification in an unguided manner. We introduce a novel guided search algorithm based on MCMC sampling.

**Results:** Our experimental results for the 50 buggy versions of 5 real-world subjects (i.e., Libtiff, PHP, GMP, Gzip, and Wireshark) show that our method named FAngelix is on average an order of magnitude faster than Angelix (a state-of-the-art constraint-based program repair tool), showing up to 23 times speed-up. This speed-up is achieved without sacrificing repairability and patch quality.

**Conclusion:** This paper proposes a novel technique that expedites constraint-based program repair, using a search-based technique based on MCMC sampling. Our experimental results show the promise of our approach.

## 1. Introduction

Automatic program repair has made major strides in the last decade showing exciting potential. Numerous approaches have been proposed to fix more bugs more precisely.[1] In this paper, we investigate another important and relatively less studied aspect of program repair, that is, the runtime of repair. Why runtime? A report from Facebook [1] sharing the experience of deploying Infer [2], a well-known static analyzer, provides an interesting lesson which we paraphrase here. Developers at Facebook initially paid almost zero attention to the faults reported by Infer. Only after Infer started to report faults more immediately did the fix rate "rocket to over 70%".

The runtime efficiency, despite its importance, is a relatively understudied problem in the automated program repair research [3]. The widely adopted generate-and-validate (G&V) approach repeatedly generates patch candidates until a patch is found or a timeout occurs. As reported in [3], state-of-the-art G&V tools such as SimFix [4] and TBar [5] repeatedly apply hundreds to thousands of patch candidates before being able to find a patch.

Meanwhile, the constraint-based approach (also known as the semantics-based approach) takes a different approach to generate a repair. Unlike the generate-and-validate approach, the constraint-based approach first extracts a repair constraint $\varphi$, and then synthesizes a patch satisfying $\varphi$. Given that $\varphi$ is used as a specification of a patch, a repair constraint $\varphi$ can also be viewed as a patch specification. In this paper, we will use these two terminologies, "repair constraint" and "patch specification", interchangeably.

Since a patch is synthesized in a correct-by-construction manner (rather than compiling and testing each repair candidate source code), the constraint-based approach, in theory, requires less runtime overhead than the G&V approach. Nevertheless, the performance of existing constraint-based approaches is still suboptimal. For example, it takes more than 2 h for Angelix, a state-of-the-art constraint-based repair tool, to fix a buggy Libtiff version (L-3b848). As will be shown, the same bug can be fixed two times faster with our approach.

**Reasons for suboptimal performance.** We identify two problems, one general problem and another problem specific to Angelix. The first problem stems from the fact that the existing constraint-based approaches use symbolic execution to extract a repair constraint; a suspicious expression is replaced with a symbol $\alpha$, and various execution paths are executed using symbolic execution with an aim to find

* Corresponding author.
*E-mail addresses:* jooyong@unist.ac.kr (J. Yi), elkhan@unist.ac.kr (E. Ismayilzada).

[1] We describe some of them in Section 8.

https://doi.org/10.1016/j.infsof.2022.106865
Received 25 February 2021; Received in revised form 20 January 2022; Accepted 25 January 2022
Available online 9 February 2022

an "angelic path" in which a given test passes. Then, the constraint about $\alpha$ to follow the discovered angelic path is extracted into a patch specification. While searching for an angelic path, a symbolic execution tool such as KLEE [6] blindly tries out various symbolic execution paths without knowing which one is more likely to be an angelic path than the others. In other words, *no guidance is given to symbolic execution while searching for an angelic path.*

The second problem is specific to ANGELIX, which collects multiple angelic paths instead of terminating the search as soon as the first angelic path is found. This is to avoid synthesizing an incorrect patch when a wrong angelic path is discovered before a correct one. Note that there can be many angelic paths passing a given test. ANGELIX mitigates this problem by passing a set of angelic paths to its patch synthesizer, which chooses to use an angelic path that leads to a repair syntactically closest to the original program. *However, finding out multiple angelic paths inevitably slows down the runtime performance.*

**Our guided approach.** In this work, we propose a novel guided algorithm to search for an angelic path efficiently. Our algorithm guides the search toward execution paths whose involved costs become smaller, using a well-known MCMC (Markov Chain Monte Carlo) sampling technique [7]. The cost of an execution path is computed using a user-provided cost function. While our guided algorithm is not tied to a specific cost function, we in this study evaluate our approach with simple cost functions for the sake of usability. For example, suppose a given test specifies its expected output as a string value. In that case, we return the edit distance between the expected output and the actual output of execution as a cost.

Our algorithm stops as soon as the first angelic path $\pi$ is found instead of collecting multiple angelic paths. Since $\pi$ may not be a correct angelic path, we refine $\pi$ into another angelic path $\pi'$. That is, following minimal repair heuristics [8–10] which opts for a repair syntactically or semantically closest to the original program under repair, we transform $\pi$ into $\pi'$ closest to the execution path observed in the original program. Note that we represent an execution path with a sequence of bits (0 and 1 represent the true and false branch, respectively).

**Our results.** Our experimental results for the 50 buggy versions of 5 real-world subjects (i.e., LIBTIFF, PHP, GMP, GZIP, and WIRESHARK) show that *our method named* FANGELIX *is on average 3.5 times faster than* ANGELIX*, showing up to 23 times speed-up*. Note that this speed-up is achieved without sacrificing repairability. In fact, FANGELIX succeeds in generating a patch in four more versions than ANGELIX, although our approach is not designed to improve repairability. More surprisingly, FANGELIX generated correct patches in three more versions than ANGELIX. All in all, FANGELIX *generates correct patches more frequently, consistently, and quickly than* ANGELIX.

**Our contributions.** Overall, we make the following contributions:

1. We identify why the runtime performance of the existing constraint-based repair approaches is suboptimal.
2. We introduce a novel guided specification inference technique based on MCMC sampling.
3. We provide experimental results that show that our technique, FANGELIX, outperforms ANGELIX, in terms of runtime efficiency and efficacy in generating correct patches.

Our tool and experimental results are available in the following URL: https://github.com/jyi/fangelix.

## 2. Background

In this section, we briefly describe the techniques and theoretical background upon which our approach is built. These include constraint-based program repair (Section 2.1), Markov Chain Monte Carlo sampling (Section 2.2), and delta debugging (Section 2.3).

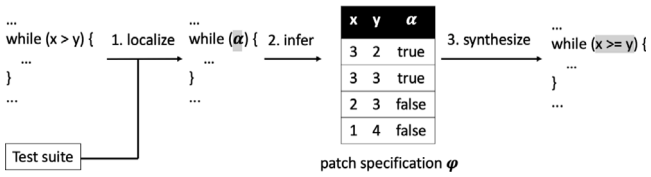



**Fig. 1.** The workflow of the constraint-based program repair. First, a suspicious location x > y is localized typically using spectrum-based fault localization. Second, a patch specification $\varphi$ for the suspicious location is inferred. Lastly, a patch, x ≥ y, satisfying $\varphi$ is synthesized.

### 2.1. Constraint-based program repair

Fig. 1 shows the workflow of constraint-based program repair. In the first step, a list of suspicious expressions is obtained, typically using spectrum-based fault localization [11], which computes the suspiciousness scores of program entities such as statements and expressions based on program spectra—runtime data collected while running a given test suite. Spectrum-based fault localization is widely used in all test-based program repair approaches. Note that the existing constraint-based program repair performs a fix at an expression granularity, and hence fault localization is performed at the expression level. In Fig. 1, expression x > y is considered to be faulty.

In the second step, a suspicious expression is replaced with a symbolic variable (denoted with $\alpha$ in Fig. 1), and subsequently, symbolic execution is performed. The goal in this step is to infer a patch specification that describes when a given test passes. For example, the table of Fig. 1 represents a patch specification. That is, a given test passes if $\alpha$ takes true, false, and false in sequence. In other words, sequence [true, false, false] forms an angelic path. The table also shows the environment (values of in-scope variables) in which an angelic path is taken. For example, when $\alpha$ is first encountered, variables x and y have 3 and 2, respectively. This should be interpreted as follows. When x and y have 3 and 2, a patched new expression should return true. The remaining three rows of the table should be interpreted similarly.

In the last step, a patch is generated using a program synthesis technique, which synthesizes a patch satisfying a given patch specification. In our running example, a synthesized expression x ≥ y satisfies all 4 rows of the patch-specification table. The existing work uses various program synthesis techniques such as component-based program synthesis [12] and syntax-guided synthesis [13]. In general, multiple expressions satisfy a patch specification, and the existing approaches use heuristics to choose one among them. For example, ANGELIX synthesizes a syntactically closest patch (in terms of the number of AST-level edits) to the original expression.

**Patch specification inference.** We describe in more detail the patch-specification-inference step, the topic of this work. Algorithm 1 shows the patch specification algorithm of ANGELIX, which we modify in this work. Given a set of suspicious expressions $E$ identified using spectrum-based fault localization, Algorithm 1 performs controlled symbolic execution. That is, during symbolic execution, every occurrence of $e$ is replaced with a fresh symbol (see line 6). The resultant path condition $pc$ is conjoined with $O_a = O_e$ where $O_a$ and $O_e$ refer to the actual output and the expected output, respectively. An angelic path leading to the expected output can be obtained by checking the satisfiability of the augmented path condition $R$ (see lines 8–11). This process is repeated until all paths are explored, or the maximum number of paths reaches (see line 3). The output of the algorithm is an angelic forest defined in Definition 1.

**Algorithm 1** The specification inference algorithm of Angelix

**Input:** program $P$
**Input:** test case $(I, O_e)$ // $I$: input, $O_e$: expected output
**Input:** a set of suspicious expressions $E$
**Output:** angelic forest $A$

```
A ← ∅; k ← 0
for e ∈ E do
    while there is an unexplored path ∧ k++ < max_k do
        /* Using a symbolic execution, extract */
          * an actual output O_a and a path condition pc */
        pc, O_a ← ControlledSymExe(P, I, e)
        R ← pc ∧ O_a = O_e
        if R is satisfiable then
            /* Extract a model using a constraint solver */
            M ← GetModel(R)
            A ← A ∪ ExtractAngelicPath(M)
        end if
    end while
    if A ≠ ∅ then
        break
    end if
end for
return A
```

```
{T:
  [L: {[({x=3, y=2}, true), ({x=3, y=3}, true),
        ({x=2, y=3}, false), ({x=1, y=4}, false)],
       [({x=4, y=2}, true), ({x=6, y=3}, false),
        ({x=2, y=3}, false)]}
  ]
}
```

**Fig. 2.** An example of an angelic forest for a test $T$ and a suspicious location $L$.

**Definition 1** (*Angelic Forest*). *An angelic forest is a singleton dictionary for a suspicious expression e*[2]*:*

$$\{e : \{[(\sigma_i, v_i)]_{i=1}^{L(j)}\}_{j=1}^{n}\}$$

*Given an angelic forest A, A[e] returns a set of angelic paths, where in each angelic path, a given test passes. Notation* $(\sigma, v)$ *denotes a pair of an angelic value* $v$ *and its environment* $\sigma$ *of* $v$*, where* $\sigma$ *consists of in-scope variables at location* $e$ *and their values at the time when* $v$ *is obtained. Notation* $[(\sigma_i, v_i)]_{i=1}^{L(j)}$ *denotes a* $L(j)$*-length sequence of pairs where* $L(j)$ *denotes the length of the* $j$*th angelic path. Lastly,* $n$ *denotes the total number of angelic paths in an angelic forest.*

Fig. 2 shows an example of an angelic forest. Notations $T$ and $L$ represent respectively the test under consideration and the location of a suspicious expression (e.g., x > y of Fig. 1). The example angelic forest contains two angelic paths where the first angelic path is a sequence of true, true, false, and false whereas the second angelic path is a sequence of true, false, and false. This means that test $T$ passes when one of these angelic paths is taken. Since it is generally not known which angelic path in an angelic forest is a correct one, the patch synthesizer of Angelix considers all available angelic paths simultaneously and produces a patch that is syntactically closest to the original buggy expression, using a MaxSMT solver. The angelic forest also contains the environment of each angelic path, the values of in-scope variables x and y which can appear in a synthesized expression.

### 2.2. Markov Chain Monte Carlo sampling

To guide the search for an angelic path, we use MCMC (Markov Chain Monte Carlo) sampling [7] in this work instead of symbolic execution. MCMC sampling is a well-established subject, and here we only provide a brief description. More information is available in [7].

In MCMC sampling, samples associated with higher probabilities are more frequently sampled than those associated with lower probabilities. In our work, a sample is viewed as an execution path and expressed as a bit-vector. For example, the first angelic path shown in Fig. 2 (i.e., true, true, false, and false) is represented with a bit-vector 1100. Using MCMC sampling, we explore the space of bit-vectors. Note that as the length of a bit-vector increases, the size of the bit-vector space increases exponentially, and MCMC sampling helps explore large space efficiently.

Among various MCMC sampling methods, we use a widely used Metropolis–Hastings algorithm [14,15]. The algorithm essentially performs a random walk by repeatedly proposing a new sample $S^*$ by mutating the current sample $S$. In our case, we repeat to randomly mutate the current execution path (represented with a bit-vector) to obtain the next execution path to consider.

An important property of MCMC sampling is that a new proposal $S^*$ is not always accepted as the next current sample. Instead, $S^*$ is accepted with the *Metropolis–Hastings acceptance probability*, $\alpha(S \to S^*)$, defined as follows:

$$\alpha(S \to S^*) = \min\left(1, \frac{p(S^*)}{p(S)} \cdot \frac{q(S|S^*)}{q(S^*|S)}\right) \tag{1}$$

where $p(S)$ and $p(S^*)$ refer to the probability density function of $S$ and $S^*$, respectively, and $q(S^*|S)$ refers to the conditional probability of proposing $S^*$ given $S$. Notation $q(S|S^*)$ can be described similarly.

The density function $p(S)$ appearing in the acceptance probability can be described with an arbitrary cost function $c$ for $S$ in the following way [16]:

$$p(S) = \frac{1}{Z}\exp\left(-\beta \cdot c(S)\right), \tag{2}$$

where $\beta$ is a configurable annealing constant and $Z$ refers to a partition function that normalizes the distribution. Combining Eqs. (1) and (2) leads to the following equation we use in this work:

$$\alpha(S \to S^*) = \min\left(1, \exp\left(-\beta \cdot k\right) \cdot \frac{q(S|S^*)}{q(S^*|S)}\right), \tag{3}$$

where $k = c(S^*) - c(S)$. Note that $Z$ of Eq. (2) is canceled out and disappears in Eq. (3). Notice that to compute $\alpha(S \to S^*)$, we need the cost of each proposal and the two conditional probabilities, $q(S^*|S)$ and $q(S|S^*)$.

### 2.3. Minimal repair and delta debugging

There can be multiple angelic paths that can pass a given test, and an angelic path found via MCMC sampling may not be a correct one. To address this issue, we use minimal repair heuristics used in previous work [8–10]. The key idea of minimal repair heuristics is that when there are multiple repairs, a repair closest to the original program is more likely to be correct than the others. For example, Angelix produces a patch syntactically closest to the original buggy expression, as mentioned in Section 2.1. Meanwhile, we apply minimal-repair heuristics at the specification level. Recall that we represent an angelic path with a bit-vector. Thus, our goal is to find a bit-vector $\vec{b}$ that satisfies the following two conditions:

**(C1)** A given test should pass with $\vec{b}$, and

**(C2)** $\vec{b}$ is closest to the bit-vector corresponding to the original execution path obtained from the buggy version.

We achieve our goal using delta debugging [17]. Algorithm 2 shows the *ddmin* algorithm of delta debugging applied to our context. Given an angelic path represented with a bit-vector $\vec{b}$ and the original execution path represented with $\vec{b}_1$, the difference between these two

[2] We assume that suspicious expressions are investigated one by one.

**Algorithm 2** $ddmin(\Delta, n)$

**Input:** a list of elements $\Delta$
▷ When $ddmin$ is initially called, $\Delta = |\vec{b} - \vec{b}_1|$ where $\vec{b}$ and $\vec{b}_1$ represent an angelic path and the original execution path, respectively. Note that a given test passes in an angelic path (i.e., $test(\vec{b}) = \checkmark$) while the same test fails for the original path (i.e., $test(\vec{b}_1) = \times$).
**Input:** a number of partitions $n$
▷ $\Delta$ is divided into $n$ partitions
**Output:** a list of elements $\Delta_{min} \subseteq \Delta$ satisfying the following:
▷ $test(update(\vec{b}_1, \Delta_{min})) = \checkmark$
▷ $\forall \delta_i \in \Delta_{min}.test(update(\vec{b}_1, \Delta_{min} - \{\delta_i\})) = \times$
**while** $test(update(\vec{b}_1, \Delta_i)) = \checkmark$ **do** ▷ $\Delta = \Delta_1 \cup \cdots \cup \Delta_n$
  /* reduce to subset */
  **return** $ddmin(update(\vec{b}_1, \Delta_i), 2)$
**end while**
**while** $test(update(\vec{b}_1, \nabla_i)) = \checkmark$ **do** ▷ $\nabla_i = \Delta - \Delta_i$
  /* reduce to complement */
  **return** $ddmin(update(\vec{b}_1, \nabla_i), max(n-1, 2))$
**end while**
**if** $n < |\Delta|$ **then**
  /* increase granularity */
  **return** $ddmin(\Delta, min(|\Delta|, 2n))$
**else**
  **return** $\Delta$
**end if**

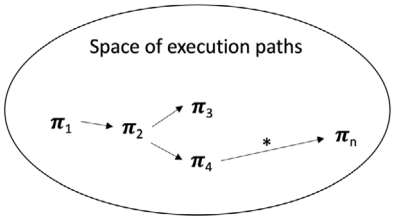



**Fig. 3.** High-level description of our approach. FAngelix searches for an angelic path $\pi_n$ via random walk starting from the initial execution path $\pi_1$. Notation $\overset{*}{\rightarrow}$ represents the transitive closure of transitions. The cost of $\pi_3$ is higher than that of $\pi_2$, and $\pi_3$ is discarded with a high probability.

bit-vectors is represented with $\Delta = |\vec{b} - \vec{b}_1|$. For example, if $\vec{b}_1$ and $\vec{b}$ are 1000 and 0001, respectively, then $\Delta$ is defined as the following two changes: (1) changing the first bit of $\vec{b}_1$ from 1 to 0 and (2) changing the last bit of $\vec{b}_1$ from 0 to 1. The $ddmin$ algorithm uses a classical "divide and conquer" approach and returns $\Delta_{min} \subseteq \Delta$ satisfying the following two conditions:

1. $test(update(\vec{b}_1, \Delta_{min})) = \checkmark$
2. $\forall \delta_i \in \Delta_{min}.test(update(\vec{b}_1, \Delta_{min} - \{\delta_i\})) = \times$

We use notation $update(\vec{b}, \Delta)$ to represent a bit-vector obtained by updating $\vec{b}$ with $\Delta$. For example, $update(1000, \{$changing the first bit to 0, changing the last bit to 1$\})$ returns 0001. Notations $\checkmark$ and $\times$ represent test success and failure, respectively. By the first condition, $\Delta_{min}$ induces an angelic path when applied to $\vec{b}_1$. And by the second condition, there is no $\Delta' \subset \Delta_{min}$ that induces an angelic path when applied to $\vec{b}_1$. Notice that these two conditions match the two aforementioned conditions C1 and C2.

## 3. Our Approach: FAngelix

Our approach is an extension of a constraint-based program repair approach, Angelix [8], which takes the following three steps to generate a repair: (1) fault localization, (2) patch specification inference, and (3) patch synthesis. Other constraint-based approaches [10,18–21] work similarly. *We change the module for patch specification inference* while reusing the existing modules of Angelix for the remaining two steps. Algorithm 3 describes our patch specification inference algorithm, replacing the counterpart of Angelix shown in Algorithm 1. A detailed description of the algorithm will be provided shortly in this section.

Fig. 3 describes a high-level overview of our approach. FAngelix searches for an angelic path $\pi_n$ via random walk starting from the initial execution path $\pi_1$. Recall that a previously failing test passes if the modified program follows an angelic path. The next execution path to explore is prepared by randomly mutating the current execution path, which is initially set to $\pi_1$. For each execution path $\pi_i$, its cost $c(\pi_i)$ is measured. If $c(\pi_i)$ is lower than that of the current execution path, the current execution path is updated into $\pi_i$. Conversely, if $c(\pi_i)$ is higher than that of the current execution path, $\pi_i$ is discarded with a probability $p$; note that it is still allowed that the current execution path is updated into $\pi_i$ with probability $1 - p$, following MCMC sampling (see Section 2.2), which prevents the search from getting stuck in local minima. Following equation (3), the probability $p$ can be defined as $1 - \alpha(\pi_c \rightarrow \pi_i)$ where $\alpha(\pi_c \rightarrow \pi_i)$ represents the probability of updating the current execution path from $\pi_c$ to $\pi_i$. Notice that $p$, the probability to discard $\pi_i$, increases as $c(\pi_i) - c(\pi_c)$ increases.

**Technical challenge.** We compute $\alpha(\pi_c \rightarrow \pi_i)$ using Eq. (3) we copy below.

$$\alpha(S \rightarrow S^*) = \min\left(1, \exp(-\beta \cdot k) \cdot \frac{q(S|S^*)}{q(S^*|S)}\right),$$

where $q(S^*|S)$ and $q(S|S^*)$ represent the conditional probability of proposing $S^*$ given $S$ and vice versa, respectively. If we assume that the size of $S$ and $S^*$ is the same (in our context, $S$ and $S^*$ are bit-vectors representing execution paths) and the bits of $S$ are mutated in a uniform manner, computing $q(S^*|S)$ and $q(S|S^*)$ is straightforward. For example, the probability of changing configuration 00 to a new configuration 01 is simply 1/4.

However, neither assumptions hold in our case. The first assumption (the size of $S$ and $S^*$ is the same) does not hold because depending on an execution path, the suspicious expression under consideration may be executed a different number of times. Thus the length of a bit-vector may vary at each execution. The second assumption (the bits of a configuration are mutated uniformly) does not hold either in Algorithm 3. Given a bit-vector $\vec{b}$ defined in the current configuration, our guided search algorithm allows both small perturbations of $\vec{b}$ (i.e., changing an arbitrary single bit of $\vec{b}$) and larger perturbations of $\vec{b}$. The former is to exploit the current configuration, and the latter is to explore other possible configurations. The combination of these (the size of a configuration and the degree of perturbation may vary) makes it non-trivial to compute $q(S^*|S)$ and $q(S|S^*)$. In this section, we show how we address this challenge.

Like Angelix, FAngelix can fix two different kinds of defect classes: conditional bugs (where a bug fix requires changing conditional expressions) and assignment bugs (where a bug fix requires changing the right-hand-side expression of an assignment statement). In the rest of this section, we describe how we fix conditional bugs (Section 3.1) and assignment bugs (Section 3.2). We also describe optimization techniques used in our implementation (Section 3.3).

### 3.1. Conditional bugs

To fix a conditional bug, Angelix and FAngelix change the original Boolean values of the conditional branches in an attempt to find an angelic path. Unlike Angelix, which uses symbolic execution to find all angelic paths that can be found in a given bound, FAngelix uses MCMC sampling to search for a single angelic path. More specifically, for a given suspicious program location containing a conditional expression $e$, we first monitor which branch directions are executed at that program location whenever $e$ is executed during program execution. We store this sequence of Boolean values representing branch directions

**Algorithm 3** The specification inference algorithm of FAngelix

**Input:** program $P$
**Input:** test case $(I, O_e)$ // $I$: input, $O_e$: expected output
**Input:** a sorted set of locations $L$ of suspicious conditional expressions
**Input:** cost function Cost
**Output:** an angelic path $\pi$

```
1: /* Extract the original output O_1 and */
2:   * the value trace of E denoted with S⃗ */
3: O_1, S⃗ ← Run(I, L)
4: C_1 ← Cost(O_1, O_e) // C_1: initial cost
5: for l ∈ L do
6:     /* Phase 1: Initialization */
7:     S ← S⃗[l] // S: initial configuration for l
8:     O, C ← O_1, C_1
9:     N ← 0 // N: the number of trials
10:    /* Phase 2: Search for an angelic path */
11:    while O ≠ O_e ∧ Continue(N, C) do
12:        /* perform MCMC sampling */
13:        S′ ← Propose(S)
14:        O, S* ← Run(I, S′)
15:        C* ← Cost(O, O_e)
16:        if Accept(C, C*) then
17:            S, C ← S*, C*
18:        end if
19:        N ← N + 1
20:    end while
21:    /* Phase 3: Post-processing */
22:    if O = O_e then
23:        S_1 ← S⃗[l] // the original buggy configuration for l
24:        S† ← Refine(S, S_1)
25:        return S†
26:    end if
27: end for
28: return ⊥ // inference failure
```

```
while (wrapper(/* the original expression: */
               x > y,
               /* location of expression: */
               "10–7–10–11",
               /* names of environment variables: */
               ((char*[]){"x", "y"}),
               /* values of environment variables: */
               ((int[]){x, y}),
               /* the number of environment variables: */
               2)) { ... }
```

**(a) An example of using a wrapper function for a suspicious expression `x > y`**

```
{"10–7–10–11": [1, 1, 0, 0]}
```

**(b) An example of a configuration for location 10-7-10-11 where the four numbers describe the starting line number, the staring column number, the ending line number, and the ending column number of the location, respectively**

**Fig. 4.** An example of a wrapper function and a configuration.

as a bit-vector (1 representing true and 0 representing false). We then randomly mutate the current bit-vector (e.g., 1110 can be mutated to 1010) and store it as a configuration file. In the next program execution, we force the execution path to follow the branch directions as specified in the configuration file.

For each execution, we record the involved cost using a user-provided cost function, and use this cost to guide the search for an angelic path. More specifically, if the cost obtained using a new configuration $S$ is smaller than the cost obtained last, we update the current configuration to $S$ so that $S$ can be mutated next time. Otherwise, we discard $S$ with a high probability $p$ or accept $S$ as the next configuration with probability $1 - p$.

There is no need to use symbolic execution to implement our MCMC-based approach. The sample space is restricted to a set of bit-vectors, and we explore the sample space using fast native execution in the following way.[3] We transform a given buggy program to wrap each suspicious expression with a wrapper function. Fig. 4(a) shows an example. During runtime, a wrapper function returns a value as specified in its matching configuration. As an example, the configuration shown in Fig. 4(b) specifies that the suspicious expression $e$ located at location 10-7-10-11 should return 1, 1, 0, and 0 in sequence, one by one, as $e$ is encountered during execution. In the following subsections, we provide a more detailed description of our algorithm.

### 3.1.1. Configurations

We search for an angelic path by repeatedly mutating configurations. We define a configuration as a singleton dictionary $\{l : [0,1]^+\}$ where $l$ represents the program location for a suspicious conditional expression, and $[0,1]^+$ denotes a bit-vector whose length is at least 1. Fig. 4(b) shows an example of a configuration. The $i$th bit of the bit-vector in a configuration dictates which Boolean value should be used during runtime at the $i$th occurrence of suspicious conditional expression at location $l$.

As an initial configuration, we use the configuration observed when running the original buggy program $P$ with a given test input $I$. While running $P$ with $I$, we monitor and record which branch is taken at each evaluation of a suspicious expression. Note that a set of locations $L$ for suspicious conditional expressions is part of the input to Algorithm 3. FAngelix obtains $L$ by performing statistical fault localization, and $L$ is sorted in a descending order based on the suspiciousness scores of the locations in $L$.

### 3.1.2. Overall procedure

The loop between line 5 and 27 describes the main procedures of our algorithm and consists of the three phases. In the first phase (lines 7–9), we initialize variables including $S$ (which holds the current configuration), $O$ (which holds the current output), $C$ (which holds the current cost), and $N$ (which holds the number of trials). Variables $S, C$, and $O$ are initialized with the configuration, output, and cost obtained from the execution of the original buggy program.

In the second phase, we search for an angelic path via MCMC sampling. In the while loop between lines 11 and 20, we repeat to perform the following. First, we mutate the current configuration $S$ into $S'$ by calling Propose (line 13). Then, we run the program with $S'$ and obtain output $O$ with which we compute the cost $C^*$ associated with $O$. Lastly, we decide whether to update the current configuration by calling Accept (line 16).

We perform the last phase if an angelic path is found. Note that in an angelic path, the obtained output $O$ equals the expected output $O_e$. We post-process the obtained angelic path by calling Refine (line 24), which performs delta debugging to find an angelic path closest to the initial configuration. In the remaining, we describe each step in more detail.

[3] The comparison between the expected and actual output is performed directly by running tests natively, unlike in Angelix where the comparison is made via symbolic execution.

#### 3.1.3. Proposing the next configuration

The Propose function in line 13 proposes the next configuration $S'$ by mutating the current configuration $S$. Given $S$ defined as $\{l : \vec{b}\}$ where $l$ and $\vec{b}$ represent the location of the suspicious expression and its bit-vector, respectively, we choose either to flip one bit of $\vec{b}$ with a probability $p_1$, or to flip random $n$ bits of $\vec{b}$ where $1 < n \leq |\vec{b}|$ with the probability $1 - p_1$. In our experiments, we use 0.5 for $p_1$. This design is to give a higher probability for a one-bit flip than other numbers of flip while still allowing to flip more than one bit. When it is chosen to flip $k$ bits, we choose $k$ bits of $\vec{b}$ in a uniform manner. For example, if the size of a bit-vector is $n$, then each bit has a chance of a flip with probability $k/n$.

#### 3.1.4. Dynamic adjustment of a configuration

When the next configuration $S'$ is generated by mutating $S$, the length of the bit-vector of $S'$, which we denote with $|S'|$, is the same as $|S|$. However, when the program under repair is executed with $S'$, the current suspicious location may be executed a different number of times than $|S'|$. We need to factor in the actual number of times the current suspicious location is executed at runtime.

A naive sampling approach would be to fix the size of a bit-vector to $N$. In this approach, only the first $k$ bits, where $k < N$, are used when the current suspicious expression is executed $k$ times, and the remaining bits are ignored. However, this approach can unnecessarily increase the size of the search space if $N$ is larger than is necessary. On the contrary, if $N$ is smaller than is necessary, the $k$th occurrences of the suspicious expression where $k > N$ cannot be controlled with a configuration.

To avoid these problems, we adjust $S'$ into $S^*$ where $|S^*|$ is the same as the actual number of times the current suspicious expression is executed at runtime. More specifically, we run the test with $S'$ and record the trace of the values of expression $e$ in the current suspicious location $l$. At runtime, the values specified in $S'$ are used in sequence until $e$ is executed $|S'|$ times. After all bit values specified in $|S'|$ are exhausted, we return a random bit value whenever $e$ is executed. In case $e$ is executed less than $|S'|$ times, the trace of $l$ only contains the bit values that are actually used. We extract an adjusted configuration $S^*$ from the runtime trace of $l$ (line 14).

Note that the ergodicity of proposals necessary for MCMC sampling is satisfied in our approach. That is, any feasible configuration corresponding to a real execution path in the search space can be obtained through a series of proposals (i.e., calling the Propose function) and test execution (i.e., calling the Run function).

#### 3.1.5. Measuring the cost of a configuration

To perform a guided search using MCMC sampling, we measure the cost of each configuration (line 15). Ideally, we want to measure the distance between the execution path associated with $S^*$ and an angelic path. However, since an angelic path is not known in advance, we instead measure a surrogate distance, using a user-provided project-specific cost function Cost. A cost function essentially measures the distance between the expected output $O_e$ with the actual output $O$ obtained when configuration $S^*$ is used.

#### 3.1.6. Updating the current configuration

Our guided search is performed through a series of updates of configurations. We update a configuration using the Metropolis–Hastings algorithm as follows. We compare the cost $C^*$ obtained when a new configuration $S^*$ is used with the current cost $C$. If $C^* < C$, we accept $S^*$ as the new current configuration. Otherwise, we reject $S^*$ and keep the current configuration $S$ with probability $p < 1$. The reason why we do not reject $S^*$ with probability 1 is to avoid being stuck in a local minimum. The rejection probability increases as the difference between $C^*$ and $C$ increases. More concretely, we use the following cost-based Metropolis–Hastings acceptance probability mentioned in Section 2.

$$\alpha(S \rightarrow S^*) = \min\left(1, \exp(-\beta \cdot k) \cdot \frac{q(S|S^*)}{q(S^*|S)}\right), \tag{4}$$

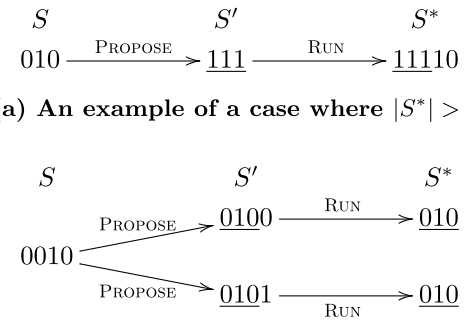



**Fig. 5.** Examples of cases where $|S| \neq |S^*|$.

where $\beta$ is a configurable annealing constant (we use 0.8 in our experiment), and $k = \text{Cost}(S^*) - \text{Cost}(S)$. Notation $\alpha(S \rightarrow S^*)$ represents the acceptance probability which dictates the probability of accepting $S^*$ as the next current configuration when the current configuration is $S$. The value of $k$ shows the difference between $\text{Cost}(S^*)$ and $\text{Cost}(S)$.

In Eq. (4), $q(S^*|S)$ denotes the probability of transforming $S$ into $S^*$ using the Propose function (where $S$ is transformed into $S'$) and the Run function (where $S'$ is transformed into $S^*$). If we assume that the sizes of the bit-vectors in $S$ and $S^*$ are equivalent to each other, and the bits of $S$ are mutated uniformly, computing $q(S^*|S)$ and $q(S|S^*)$ is straightforward. For example, the probability of changing configuration 00 to a new configuration 01 is simply 1/4. However, neither assumptions hold in our case. The first assumption (the sizes of the bit-vectors in $S$ and $S^*$ are equivalent to each other) does not hold because depending on an execution path, the suspicious expression under consideration may be executed a different number of times. The second assumption (the bits of a configuration are mutated uniformly) does not hold either in our algorithm, as described in Section 3.1.3. Given a bit-vector $\vec{b}$ defined in the current configuration, our guided search algorithm allows both small perturbations of $\vec{b}$ (i.e., changing an arbitrary single bit of $\vec{b}$) and larger perturbations of $\vec{b}$.

We compute $q(S^*|S)$ differently depending on various situations. First, we compute $q(S^*|S)$ as follows when $|S^*| = |S|$. Note that when $|S^*| = |S|$, it also holds that $S^* = S'$. Our Propose function chooses either to flip 1 bit with probability $p_1$ or to flip $n$ bits, where $1 < n \leq |S|$, with probability $1 - p_1$, as mentioned in Section 3.1.3. Given $S$ and $S'$ where $S' = \text{Propose}(S)$, suppose that $k$ bits, where $k > 1$, are different between $S$ and $S'$. The probability of flipping these particular $k$ bits can be computed as follows where we assume that $|S| = l$:

$$(l-1)^{-1} \cdot \binom{l}{k}^{-1} \cdot (1 - p_1), \tag{5}$$

where $(l-1)^{-1}$ is the probability of choosing number $k$ between 2 and $l$ (inclusive), the second factor, $\binom{l}{k}^{-1}$, is the probability of choosing the particular combination of the $k$ bits among $l$ bits, and the last factor, $(1 - p_1)$, is the user-defined probability of flipping more than one bit. Meanwhile, if only one bit is different between $S$ and $S'$, the probability of transforming $S$ into $S'$ is

$$l^{-1} \cdot p_1, \tag{6}$$

where $l^{-1}$ represents the probability of flipping the particular one bit whose value is different between $S$ and $S'$, and $p_1$ is the user-defined probability of flipping one bit.

If $|S^*| > |S|$, this implies that the current suspicious expression is executed more number of times than is specified in $S'$ where $S' = \text{Propose}(S)$. Fig. 5(a) shows such an example where $S$, $S'$ and $S^*$ are 010, 111, 11110, respectively, and the common bits between $S'$ and

$S^*$ are underlined. In this case, the transformation probability can be computed as follows.

$$q(S^*|S) = q(S'|S) \cdot (1/2)^{(|S^*|-|S|)} \tag{7}$$

Note that $q(S'|S)$ can be computed using Eqs. (5) or (6), since $|S'| = |S|$. Meanwhile, $(1/2)^{(|S^*|-|S|)}$ represents the probability of obtaining the trailing bit sequence of $S^*$ by random bit value generation during runtime (see Section 3.1.4). In our example, the probability of generating a trailing bit sequence 10 is 1/4.

Lastly, if $|S^*| < |S|$, this implies that the current suspicious expression is executed less number of times than is specified in $S'$ where $S' = \textsc{Propose}(S)$. Fig. 5(b) shows such an example where the common bits between $S'$ and $S^*$ are underlined. In this example, there are two possibilities of $S'$, i.e., 0100 and 0101. Notice that the first three bits are 010 in common, and only the last bit is different. Considering these two possibilities, we compute the transformation probability as follows.

$$q(S^*|S) = \sum_{S' \in Ext(S^*)} q(S'|S), \tag{8}$$

where function $Ext$ extends the given $S^*$ to a set of all possible configurations $S'$ where the first $|S^*|$ bits of $S'$ is the same as $S^*$, and $|S'| = |S|$. In our running example, $Ext(010) = \{0100, 0101\}$. Recall that $q(S'|S)$ can be computed using Eqs. (5) or (6).

We compute $q(S|S^*)$ of Eq. (4) in a symmetric way to $q(S^*|S)$. Note that $q(S|S^*)$ equals $q(S^*|S)$ if $|S^*| = |S|$, and the following simplified equation is used.

$$\alpha(S \rightarrow S^*) = \min(1, \exp(-\beta \cdot k)) \tag{9}$$

#### 3.1.7. Stopping investigation

We stop investigating the current suspicious expression in either of the following three cases. First, we stop investigation when an angelic path is found, which is indicated by the observation that $O = O_e$ where $O$ and $O_e$ represent actual and expected output, respectively (see line 11). Second, we stop investigation when the current number of sampling trials $N$ exceeds the user-defined maximum number of sampling trials $N_{max}$. Lastly, we also stop investigation when no reduction in the cost is observed for the configurable number of times. This is to avoid wasting resources when no progress is observed. The Continue function of Algorithm 3 at line 11 returns false when the latter two cases happen.

#### 3.1.8. Refining an angelic configuration

One potential drawback of Algorithm 3 is that if the first found angelic path is not correct, an incorrect patch is synthesized. Note that Angelix [8] mitigates this problem by looking for multiple angelic paths, hoping that one of them is correct and the correct one is used for patch synthesis.[4] However, we do not take this approach to expedite the repair process.

To mitigate the problem, we post-process the obtained angelic configuration with the Refine function (line 24). Recall that a configuration consists of the bit-vector for a suspicious program location. Given the configuration $S$ for the discovered angelic path (we call such $S$ an angelic configuration) and the initial configuration $S_1$ obtained from the original buggy execution path, we compare $S$ and $S_1$ to find minimal bit changes required for $S_1$ to pass the given test. In other words, we look for an angelic path that is closest to the original buggy execution path, similar to the minimal repair heuristics [9,20] that is widely used in the literature of program repair.

We find minimal changes of bits using delta debugging [17]. More precisely, we look for a 1-minimal angelic configuration (see Definition 2), employing the 1-minimality of delta debugging. That is, given a 1-minimal angelic configuration $S^\dagger$ containing a bit-vector $\vec{b}^\dagger$, changing any single bit of $\vec{b}^\dagger$ at the $i$th position to the different bit value that appears at the $i$th position of the original buggy configuration makes the modified configuration non-angelic.

[4] This hypothesis does not always hold as we report in Section 5.2.

```
{"foo.c": {L1: [1, 1, 0, 0], L2: [1, 0]}}
```

**Fig. 6.** An example of a runtime trace of our customized symbolic execution of FAngelix which records Boolean values taken in program locations $L_i$.

**Definition 2** (*1-minimal Angelic Configuration*). *Given the original buggy configuration $S_1$ for a suspicious conditional expression $e$ and an angelic configuration $S^\dagger$ for $e$, we say that $S^\dagger$ is 1-minimal if* $\forall i$ *s.t.* $1 \leq i \leq L(\vec{b})$ : $\textsc{Run}(I, S^\dagger[e : \vec{b}^\dagger[i] \mapsto \vec{b}_1[i]]) \neq O_e$ *holds where $\vec{b}^\dagger$ and $\vec{b}_1$ denote the bit-vector of $S^\dagger$ and $S_1$, respectively, and $L(\vec{b})$ and $O_e$ denote the length of $\vec{b}$ and an expected output, respectively. Notation $S^\dagger[e : \vec{b}^\dagger[i] \mapsto \vec{b}_1[i]]$ denotes the modified configuration of $S^\dagger$ in which the $i$th bit of $\vec{b}^\dagger$ is replaced with $\vec{b}_1[i]$.*

If $|\vec{b}^\dagger| > |\vec{b}|$, the extra trailing bits in $\vec{b}^\dagger$ may overfit to a given test. Inspired by decision tree pruning [22,23], we prune out these trailing bits of the obtained 1-minimal angelic configuration. In our experiments, this heuristic helps with generating correct patches (see Section 5.2).

### 3.2. Assignment bugs

When fixing a conditional bug, FAngelix eliminates the use of symbolic execution by using MCMC sampling instead. This is possible because the angelic value for each occurrence of a conditional expression is restricted to either true or false. However, the angelic value for the RHS (right-hand-side) expression of an assignment statement, in general, cannot be restricted to a small number of values. For this reason, FAngelix uses symbolic execution to fix an assignment bug, as in Angelix.

To fix a suspicious assignment `x = E` where $E$ represents an RHS expression, both Angelix and FAngelix replace $E$ with a symbolic variable $\alpha$. Then, at the end of a symbolic execution path, the angelic value of $E$ can be obtained by solving a constraint, $pc \wedge O_a = O_e$, where $pc$ represents the path condition of the execution path, and $O_a$ and $O_e$ represent an actual output and an expected output, respectively. Note that $pc$ and $O_a$ may involve the injected symbol $\alpha$.

If the injected symbol $\alpha$ does not flow into the conditional expressions of the program under repair, a single symbolic execution path is executed, and a patch specification is obtained by solving $pc \wedge O_a = O_e$. However, if the injected symbol flows into the conditional expressions of the program, path splits may occur, spawning multiple symbolic execution paths. In this case, FAngelix performs differently from Angelix. While Angelix explores all symbolic execution paths within a given bound, FAngelix performs a guided search for an angelic path.

To show how FAngelix works, let us assume that an injected symbol flows into conditional expression $C_i$ in program location $L_i$ of file foo.c. Note that during symbolic execution, both branches of $C_i$ may become feasible, and FAngelix records which branch of $C_i$ is taken along a symbolic execution path. Fig. 6 shows an example where monitored values in each $L_i$ are recorded as a bit-vector. Note that FAngelix terminates symbolic execution once an execution path reaches the end of the program instead of performing backtracking and exploring other symbolic execution paths.

Using a recorded bit-vector as a configuration, FAngelix performs a guided search in the same way a conditional bug is handled using Algorithm 3. More specifically, FAngelix randomly mutates the current configuration and uses it in the next session of the customized symbolic execution. When the suspicious branch location is executed, FAngelix takes branches as specified in the configuration during symbolic execution. Also, the cost of each symbolic execution path is measured

to decide whether to update the current configuration, as done for conditional bugs. The only major difference from how conditional bugs is handled is that constraint $pc \wedge O_a = O_e$ is extracted using symbolic execution, which is used to extract angelic values for the suspicious RSH expression, as done in Angelix.

### 3.3. Implementation and optimization

We implement our guided specification inference algorithm on top of PyMC3.[5] We plug our MCMC-sampling-based implementation into Angelix by replacing its specification inference module with ours. To repair the RHS expressions of assignments in a guided fashion, we also customize KLEE. Our implementation contains the two optimization techniques described in the following.

#### 3.3.1. All-in-one wrapper functions

Given a suspicious expression $e$, we wrap $e$ in a wrapper function $w$, as shown in Fig. 4(a). A wrapped expression $w(e, \ldots)$, where the ellipsis represents additional values we pass to $w$ such as location information, should behave differently at each phase of repair. During the fault localization phase, $w(e, \ldots)$ should return the value of $e$, while at the same time, the location information of $e$ should be recorded to perform statistical fault localization. During the specification inference phase, $w(e, \ldots)$ should return values as specified in the configuration of $e$ in case $e$ is a conditional expression. If $e$ is the RHS expression of an assignment, $w(e, \ldots)$ should return a fresh symbol to perform symbolic execution as described in Section 3.2. One possible way to support multiple behaviors of a wrapper function is to transform the original buggy file differently at each repair phase, as done in Angelix. To avoid multiple compilations followed by multiple program transformations, we combine all required functionalities of a wrapper function into a single wrapper function and perform program transformation only once.

#### 3.3.2. Caching and bit operations

While randomly generating a series of configurations using the Propose function, the same configurations may be generated. To avoid duplicate test execution, we cache test results and their associated costs. Also, to store a bit-vector efficiently, we maintain bit-vectors using integers. For example, a bit-vector 0101 is represented with a pair (5, 4) where 5 is the decimal number equivalent to 0101 and 4 is the number of binary digits required for the bit-vector. We perform the manipulation and comparison of bit-vectors through efficient bit operations.

## 4. Experimental setup

In this section, we describe our research questions (Section 4.1), experimental subjects (Section 4.2), experimental setup (Section 4.3), and defect classes we consider (Section 4.4). We also describe how we determine the correctness of patches (Section 4.5) and how we define cost functions (Section 4.6).

### 4.1. Research questions

Given that our method, FAngelix, replaces the patch specification inference module of Angelix in an attempt to speed up the repair process, we evaluate FAngelix in comparison with Angelix. More recent tools using constraint-based approaches such as S3 [10] and SOSRepair [21] either repair Java programs [10] (FAngelix repairs C programs) or run slowly (as mentioned in [21], "efficiency is not a focus of SOSRepair's design"), and we do not compare with them in this work. In this work, we ask the following six research questions.

**RQ 1 (Repairability).** Does FAngelix generate as many patches as Angelix? Note that unlike Angelix, FAngelix favors early termination over exhaustive search when no progress is observed (see Section 3.1.7). Does this policy negatively affect the overall repairability?

**RQ 2 (Patch quality).** Does FAngelix produce *correct* patches as often as Angelix? Note that while we reuse the patch synthesis module of Angelix, a patch specification obtained from our method can be different from that of Angelix, which may affect the quality of synthesized patches.

**RQ 3 (Efficiency).** Does FAngelix generate patches faster than Angelix? This is the primary research question of this work.

**RQ 4 (Early failure).** When a patch is failed to be generated, does FAngelix report repair failure faster than Angelix? It is desirable for a repair tool to terminate as early as possible, if the tool cannot generate a patch for a given buggy version.

**RQ 5 (Cost functions).** How effective are our cost functions in terms of performing a search for patch specification? Does the use of our cost functions improve search efficiency?

**RQ 6 (Comparison with the G&V approach).** How is the runtime performance of FAngelix compared to a G&V tool? While our main objective in this work lies in speeding up constraint-based program repair, it would be worthwhile to compare the runtime performance of FAngelix with that of the G&V approach.

### 4.2. Experimental subjects

We use the following criteria to collect experimental subjects.

**C1.** Experimental subjects should contain real-world bugs.

**C2.** Experimental subjects should be able to be run by FAngelix and Angelix. Both tools use KLEE [6], which currently does not support arbitrary C projects.

**C3.** The bugs of the experimental subjects should be in the defect class of FAngelix and Angelix. Both tools can handle the same defect classes, i.e., conditional bugs and assignment bugs.

We extract our subjects from the ManyBugs benchmark [24], part of the BugZoo [25] platform. The same benchmark has been used in multiple previous works [8,21,26–30]. The ManyBugs benchmark consists of 159 buggy versions of 7 kinds of programs[6] containing real bugs, which satisfies C1. Among the 7 kinds of programs, we exclude Python and Lighttpd since we could not run them with KLEE (see C2). These subjects are also excluded in previous work on Angelix for the same reason [8]. To meet C3, we use the previous work on Angelix [8] which identified 32 bugs in the defect classes of Angelix, and our experimental subjects include all of them. We relax C3 and add 18 more versions randomly selected from the ManyBugs benchmark. While Angelix and FAngelix fail to generate patches for these additional versions, we use them to answer RQ4 (RQ on early failure).

Table 1 summarizes the 50 versions of the 5 different subjects we used for the experiments.

[5] https://docs.pymc.io/.

[6] fbc is excluded in the latest ManyBugs benchmark since it cannot be run on a modern 64-bit Linux system.

**Table 1**
Experimental subjects.

| Subject | LoC | Tests | Versions |
|---|---|---|---|
| Wireshark | 2814k | 63 | 5 |
| PHP | 1046k | 85 | 21 |
| Gzip | 491k | 12 | 4 |
| GMP | 145k | 146 | 2 |
| Libtiff | 77k | 78 | 18 |

### 4.3. Tool configurations and experimental setup

For a fair comparison between FAngelix and Angelix, we use the same default fault localization formula (jaccard [31]) of Angelix for both tools. Similarly, we reuse the same synthesizer options used in earlier work on Angelix [8] for both tools. Note that both tools share the same modules for fault localization and patch synthesis, and only the patch specification modules are different.

When comparing the runtime performance of FAngelix and Angelix, it is crucial to control the patch specification space of the tools in a fair manner. Note that Angelix controls patch specification space with $N_{forks}$, the maximum number of forks of KLEE (a new symbolic execution path is explored for each fork), whereas FAngelix controls the same with $N_{max}$, the maximum number of sampling trials. We use the same value for $N_{forks}$ and $N_{max}$ to assign the two tools the same patch specification space.

All our experiments were performed on Intel Xeon-Gold-6154 3.00 GHz CPU with Ubuntu 18.04 OS. Since FAngelix takes a random approach during the repair process, we report an average running time after running FAngelix with each version 10 times. For a fair comparison, we also run Angelix 10 times for each version. We set the timeout to 4 h for Angelix, which is about 4 times longer than the maximum time taken for FAngelix to generate a repair (3940 s).

To answer RQ6, we need to run a G&V repair tool over our experimental subjects. While many repair tools using the G&V approach have been introduced recently (e.g., [4,5,32–34]), most of these tools work for Java programs. Since recent repair tools such as [35] are not publicly available, and the download URL for SPR [27] and Prophet [28] was down at the time of conducting this research, we ran GenProg [26] available through the BugZoo platform [25] using its pre-configured configuration. We set the timeout as 4 h and run each version 10 times.

### 4.4. Defect classes

Both Angelix and FAngelix can fix the following two kinds of bugs: conditional bugs and assignment bugs. In our experiments, we start with the conditional-bug mode first, and only if a patch is failed to be generated, we perform repair with the assignment-bug mode. We run the conditional-bug mode before the assignment-bug mode because (1) the conditional-bug mode usually terminates faster than the assignment-bug mode, and (2) the same ordering was used in the previous study [8]. In this study, we consider only first-order repair (fixing a single program location). A study on the effect of a guided algorithm on high-order repairs (fixing multiple program locations) is left as future work.

### 4.5. Patch correctness

We consider a generated patch correct if the patch is syntactically or semantically equivalent to the developer-provided patch in the benchmark. The same approach has been used in other studies [8,27,28]. Since the patch space of FAngelix is identical with that of Angelix, we reuse the patch analysis results of Angelix.[7]

[7] https://angelix.io/patches.html.

```
--FILE--
<?php
function test_1() {
    ...
    echo "Outer function increments \$v to $v\n";
    ...
}
...
--EXPECT--
Outer function increments $v to 1
Inner function reckons $v is 1
...
```

**Fig. 7.** The snippet of PHP test bug54039 where the FILE segment and the EXPECT segment show the input and the expected output of the test, respectively.

### 4.6. Cost functions

FAngelix requires a cost function as part of input. We use either of the following two kinds of cost functions. First, when the output of a test is sizeable (an exit code such as 0 is not considered sizeable), we compute the distance between the expected output and the actual output of a test. For example, Fig. 7 shows a PHP test that receives an input PHP program (shown in the FILE segment) and compares the interpretation result of the input program with the expected output (shown in the EXPECT segment). In PHP, we define a cost function as the string distance (Levenshtein distance) between the actual output and the expected output specified in the test. In Gzip and Wireshark, we similarly define cost functions using string distance. In Libtiff where tests manipulate image files, we compare the distance between an obtained image output and the expected image of a test.

In GMP where tests simply output either 0 or 1, we use the second kind of cost functions. In GMP, a test consists of multiple sub-tests (for example, the reuse test consists of 27 526 sub-tests), and once a sub-test fails, the remaining sub-tests are not executed. Exploiting this fact, we define a cost function as the number of remaining sub-tests that are not executed.

In both kinds of cost functions, we add a penalty value to an obtained cost value when the program under repair crashes, in order to avoid crash-causing configurations.

## 5. Experimental results

Table 2 shows the results of our experiments. The first and second columns show the version and the tool, respectively, while the remaining columns show the experimental results, which we explain in this section. In the version column, the initial letters stand for the 5 subjects in our benchmark (**L**ibtiff, G**M**P, **P**HP, **W**ireshark, and G**Z**IP).

### 5.1. RQ1: Repairability

As shown in the third column of Table 2, FAngelix consistently generates patches (10 times out of 10 trials) from 28 versions. This result implies that our approach maintains the effectiveness of the repair, despite the use of the early-termination policy. In fact, Angelix failed to generate a patch in 4 versions (M-13421, P-fefe9, W-37171, and Z-3fe0c). In P-fefe9, Angelix failed to find an angelic forest within the timeout. In M-13421, Z-3fe0c and W-37171, Angelix succeeded in finding angelic forests, but it failed to synthesize valid patches passing all tests using the obtained angelic forests.[8]

[8] In the original Angelix experiment, patches were generated from these four versions for which customized configurations were used.

**Table 2**
Experimental results.

| Version | Tool | Patch found | Correct patch | Time (s) Median | IQR | Speedup | Version | Tool | Patch found | Correct patch | Time (s) Median | IQR | Speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L-09e82 | Angelix | 0 | 0 | 330 | 24 | ×1.45 | P-01745 | Angelix | 10 | 0 | 916 | 12 | ×4.48 |
| L-09e82 | FAngelix | 0 | 0 | 227 | 10 | | P-01745 | FAngelix | 10 | 0 | 205 | 4 | |
| L-15632 | Angelix | 10 | 0 | 564 | 4 | ×1.1 | P-0de2e | Angelix | 0 | 0 | 6 530 | 9315 | ×6.9 |
| L-15632 | FAngelix | 10 | 0 | 512 | 91 | | P-0de2e | FAngelix | 0 | 0 | 947 | 1526 | |
| L-2e8b2 | Angelix | 0 | 0 | 263 | 7 | ×1.3 | P-11941 | Angelix | 10 | 0 | 337 | 21 | ×1.73 |
| L-2e8b2 | FAngelix | 0 | 0 | 202 | 47 | | P-11941 | FAngelix | 10 | 0 | 195 | 2 | |
| L-37133 | Angelix | 10 | 10 | 1 903 | 7 | ×1.4 | P-14738 | Angelix | 0 | 0 | 676 | 590 | ×1.55 |
| L-37133 | FAngelix | 10 | 10 | 1 355 | 65 | | P-14738 | FAngelix | 0 | 0 | 436 | 599 | |
| L-3b848 | Angelix | 10 | 0 | 8 235 | 326 | ×2.18 | P-187eb | Angelix | 10 | 0 | 316 | 9 | ×1.58 |
| L-3b848 | FAngelix | 10 | 10 | 3 770 | 106 | | P-187eb | FAngelix | 10 | 0 | 200 | 3 | |
| L-60747 | Angelix | 0 | 0 | 4295 | 7681 | ×14.31 | P-1f499 | Angelix | 0 | 0 | 567 | 391 | ×1.41 |
| L-60747 | FAngelix | 0 | 0 | 300 | 173 | | P-1f499 | FAngelix | 0 | 0 | 403 | 529 | |
| L-64062 | Angelix | 0 | 0 | 14 400 | 0 | ×14.75 | P-2a696 | Angelix | 0 | 0 | 2 792 | 1973 | ×2.85 |
| L-64062 | FAngelix | 0 | 0 | 976 | 887 | | P-2a696 | FAngelix | 0 | 0 | 979 | 653 | |
| L-6746b | Angelix | 0 | 0 | 14 400 | 0 | ×16.1 | P-53204 | Angelix | 0 | 0 | 1 818 | 623 | ×11.47 |
| L-6746b | FAngelix | 0 | 0 | 894 | 375 | | P-53204 | FAngelix | 0 | 0 | 158 | 784 | |
| L-764db | Angelix | 10 | 0 | 377 | 1 | ×1.09 | P-5bb0a | Angelix | 0 | 0 | 5 106 | 6162 | ×6.55 |
| L-764db | FAngelix | 10 | 0 | 348 | 3 | | P-5bb0a | FAngelix | 0 | 0 | 779 | 1172 | |
| L-827b6 | Angelix | 0 | 0 | 1 602 | 100 | ×6.48 | P-63673 | Angelix | 10 | 4 | 451 | 56 | ×2.45 |
| L-827b6 | FAngelix | 0 | 0 | 247 | 122 | | P-63673 | FAngelix | 10 | 10 | 184 | 1 | |
| L-96a5f | Angelix | 10 | 0 | 405 | 5 | ×1.12 | P-70075 | Angelix | 10 | 0 | 434 | 40 | ×2.1 |
| L-96a5f | FAngelix | 10 | 0 | 362 | 1 | | P-70075 | FAngelix | 10 | 0 | 206 | 3 | |
| L-a72cf | Angelix | 10 | 0 | 3 600 | 217 | ×19.22 | P-793cf | Angelix | 0 | 0 | 14 400 | 0 | ×18.9 |
| L-a72cf | FAngelix | 10 | 0 | 187 | 4 | | P-793cf | FAngelix | 0 | 0 | 762 | 117 | |
| L-b2ce5 | Angelix | 0 | 0 | 548 | 273 | ×2.39 | P-8138f | Angelix | 10 | 5 | 415 | 7 | ×2.17 |
| L-b2ce5 | FAngelix | 0 | 0 | 229 | 54 | | P-8138f | FAngelix | 10 | 5 | 191 | 2 | |
| L-ce4b7 | Angelix | 0 | 0 | 3 068 | 2735 | ×4.03 | P-86efc | Angelix | 0 | 0 | 1 161 | 161 | ×2.56 |
| L-ce4b7 | FAngelix | 0 | 0 | 762 | 1290 | | P-86efc | FAngelix | 0 | 0 | 453 | 582 | |
| L-d59e7 | Angelix | 0 | 0 | 1 058 | 537 | ×1.96 | P-a6c0a | Angelix | 0 | 0 | 619 | 543 | ×1.61 |
| L-d59e7 | FAngelix | 0 | 0 | 539 | 499 | | P-a6c0a | FAngelix | 0 | 0 | 385 | 526 | |
| L-e8a47 | Angelix | 10 | 10 | 322 | 35 | ×1.16 | P-b60f6 | Angelix | 10 | 0 | 5 047 | 432 | ×23.11 |
| L-e8a47 | FAngelix | 10 | 10 | 277 | 7 | | P-b60f6 | FAngelix | 10 | 0 | 218 | 9 | |
| L-eb326 | Angelix | 10 | 0 | 482 | 7 | ×0.92 | P-d890e | Angelix | 10 | 0 | 2 758 | 71 | ×13.35 |
| L-eb326 | FAngelix | 10 | 0 | 522 | 35 | | P-d890e | FAngelix | 10 | 0 | 207 | 7 | |
| L-eec4c | Angelix | 10 | 0 | 3 624 | 186 | ×19.17 | P-e65d3 | Angelix | 10 | 10 | 1 112 | 7 | ×4.95 |
| L-eec4c | FAngelix | 10 | 0 | 189 | 2 | | P-e65d3 | FAngelix | 10 | 10 | 224 | 15 | |
| M-13421 | Angelix | 0 | 0 | 4 890 | 22 | ×4.82 | P-eb0dd | Angelix | 0 | 0 | 777 | 57 | ×1.9 |
| M-13421 | FAngelix | 10 | 10 | 1 015 | 8 | | P-eb0dd | FAngelix | 0 | 0 | 409 | 544 | |
| M-14167 | Angelix | 10 | 5 | 887 | 24 | ×1.09 | P-f912a | Angelix | 0 | 0 | 516 | 17 | ×3.28 |
| M-14167 | FAngelix | 10 | 4 | 816 | 21 | | P-f912a | FAngelix | 0 | 0 | 158 | 10 | |
| W-37111 | Angelix | 10 | 0 | 1 223 | 23 | ×1.3 | P-fefe9 | Angelix | 0 | 0 | 2 624 | 141 | ×2.41 |
| W-37111 | FAngelix | 10 | 0 | 940 | 64 | | P-fefe9 | FAngelix | 10 | 0 | 1 087 | 1812 | |
| W-37123 | Angelix | 0 | 0 | 8 241 | 160 | ×3.87 | Z-1a085 | Angelix | 0 | 0 | 1 806 | 763 | ×3.95 |
| W-37123 | FAngelix | 0 | 0 | 2 130 | 3870 | | Z-1a085 | FAngelix | 0 | 0 | 457 | 120 | |
| W-37171 | Angelix | 0 | 0 | 7 624 | 29 | ×15.18 | Z-3eb60 | Angelix | 10 | 0 | 478 | 211 | ×1.96 |
| W-37171 | FAngelix | 10 | 0 | 502 | 24 | | Z-3eb60 | FAngelix | 10 | 10 | 244 | 155 | |
| W-37173 | Angelix | 10 | 0 | 6 745 | 46 | ×14.4 | Z-3fe0c | Angelix | 0 | 0 | 8 108 | 31 | ×10.65 |
| W-37173 | FAngelix | 10 | 0 | 468 | 4 | | Z-3fe0c | FAngelix | 10 | 0 | 761 | 564 | |
| W-37285 | Angelix | 10 | 0 | 1 811 | 284 | ×3.59 | Z-a1d3d | Angelix | 10 | 0 | 507 | 340 | ×3.65 |
| W-37285 | FAngelix | 10 | 0 | 504 | 37 | | Z-a1d3d | FAngelix | 10 | 0 | 139 | 11 | |

In our experiment, FAngelix generated patches in 4 more versions than Angelix. Also, FAngelix consistently generates patches when repair succeeds.

### 5.2. RQ2: Patch quality

The fourth column of Table 2 shows how often correct patches are generated. FAngelix generated correct patches from 9 versions; 3 from Libtiff, 2 from GMP, 3 from PHP and 1 from Gzip. Fig. 8 summarizes the patch correctness results of FAngelix and Angelix. To our surprise, Angelix generated correct patches in a less number of versions (i.e., 6 versions).

This is an unexpected result, considering that our goal is to improve repair efficiency, and we reuse the patch synthesizer of Angelix. Our investigation revealed that the exhaustive search of Angelix does not always help generate a correct patch. Listing 1(a) shows an incorrect Angelix patch generated for L-3b848, whereas Listing 1(b) shows a correct patch generated from FAngelix. Note that for this buggy version, Angelix failed to generate a correct patch. What happened is that the exhaustive search of Angelix collects multiple angelic paths, which include not only a correct angelic path but also incorrect angelic paths that overfit to a given test suite. Although the patch synthesizer of Angelix synthesizes a patch that requires minimal structural changes, both patches shown in Listing 1 involve the same amount of structural

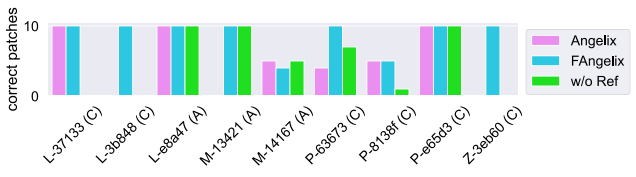



**Fig. 8.** Comparison of the number of correct patches out of 10 trials between Angelix, FAngelix, and FAngelix without using angelic configuration refinement described in Section 3.1.8. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

```
- } else if (td->td_nstrips > 1
              && td->td_compression == COMPRESSION_NONE
+ } else if (td->td_nstrips > td->td_nstrips
              && td->td_compression == COMPRESSION_NONE
```

**(a) An incorrect patch generated from Angelix**

```
- } else if (td->td_nstrips > 1
              && td->td_compression == COMPRESSION_NONE
+ } else if (td->td_nstrips > 2
              && td->td_compression == COMPRESSION_NONE
```

**(b) A correct patch generated from FAngelix**

**Listing 1.** Patches generated for L-3b848.

change (replacing one operand). This result exhibits that the exhaustive search can confuse a synthesizer by including a spurious angelic path in a patch specification. Our guided search does not exhibit the same problem because it finds only one angelic path closest to the original buggy execution path.

Recall that to find an angelic path that is closest to the original buggy execution path, we use an angelic configuration refinement method (see Section 3.1.8), and it turns out that this step is crucial. If we do not use the refinement step, a correct patch for L-3b848 is not obtained. In Fig. 8, the results of FAngelix without using the refinement step are shown with green color. Correct patches were generated only when the refinement step was used in three versions (L-37133, L-3b848, and Z-3eb60).

In Z-3eb60, Angelix failed to generate a correct patch because KLEE fails to run several positive tests that can exclude incorrect patches. Angelix ignores tests KLEE cannot run. For this version, FAngelix fixes a conditional expression using native execution without using KLEE. In M-13421, timeouts occurred for Angelix, whereas FAngelix successfully generated patches within the same timeout. Correct patches were not always generated in the three versions (M-14167, P-63673, and P-8138f) in Angelix and FAngelix because the obtained individual angelic path is not tight enough to force only one minimal patch.

> FAngelix generates correct patches more frequently than Angelix, showing the efficacy of our configuration refinement method.

### 5.3. RQ3: Efficiency

The fifth and sixth columns of Table 2 show the median time taken for each tool to terminate and the interquartile range (IQR), respectively. Note that both tools can terminate in the following three cases; (1) a repair is found, (2) the repair process is finished without finding a repair, and (3) a timeout (14 400 s) occurs. The last column of the table shows how fast FAngelix is compared to Angelix in terms of median time. FAngelix terminates faster than Angelix across all versions except for one version (L-eb326). In P-b60f6, FAngelix generates patches 23 times faster than Angelix, showing the largest speed-up. The overall median time of FAngelix and Angelix is 229 s and 719 s, respectively. And the overall mean time of FAngelix and Angelix is 513 s and 1790 s, respectively. On average, FAngelix generates a patch 3.5 times faster than Angelix.

We also show in Fig. 9 the boxplots of the running time of Angelix and FAngelix when a patch is successfully generated in FAngelix. In the x-axis, buggy versions are annotated with the defect classes of generated patches ('C' for conditionals and 'A' for assignments).

It is clear from our results that FAngelix generates patches significantly faster than Angelix. We have conducted the Mann–Whitney rank test [36] for versions repaired by FAngelix, and the running time differences are statistically significant ($p$-value < 0.001) across all versions except for three versions (L-15632, L-e8a47, L-eb326).

> FAngelix generates patches significantly faster than Angelix, showing up to x23 speed-up. On average, an order-of-magnitude speed-up is observed.

### 5.4. RQ4: Early failure

Fig. 10 shows the boxplots of the running time of Angelix and FAngelix when both tools fail to generate a patch. Across all versions, FAngelix reports repair failure faster than Angelix. Running time differences are statistically significant across all versions ($p$-value < 0.001 for the Mann–Whitney rank test) except for three versions (P-14738, P-1f499, P-a6c0a).

> FAngelix terminates the search for a patch when no progress is observed. Due to the use of this early-failure policy, FAngelix terminates earlier than Angelix, regardless of whether patches are generated or not. However, our experimental results show that the use of an early-failure policy does not decrease repairability.

### 5.5. RQ5: RQ functions

In FAngelix, the search for a patch specification is guided by a user-provided cost function by which the search performance can be affected. To obtain an initial understanding of how the cost functions we used in the experiments affect the search performance, we compare the repair time of FAngelix with its variance where we ignore the costs computed by the cost function and always accept proposed configurations. This variance can be viewed as using a simple generic cost function that monotonically decreases. Recall that during MCMC sampling, a proposed configuration whose cost is lower than the previous cost is always accepted. We keep using the same Propose function in the variance to use the same mutation operators for the configurations. We evaluate the quality of the cost functions used in our experiments by comparing the search performance between our cost functions and the monotone decreasing cost function.

Fig. 11 shows the comparison results for the 28 versions for which FAngelix generates patches. In the box plot, the "ignoreCost" label shows the results from the variance. The results for Libtiff and PHP are shown in the first two rows of the figure, and the bottom row shows the results for GMP, Gzip, and Wireshark. In Libtiff and PHP, repair time tends to decrease when our cost functions are used. This result suggests that our domain-specific cost functions are effective, performing a search more efficiently than the generic cost function. Meanwhile, in a smaller number of cases shown in the last row, repair time tends to be similar between the two modes (except in Z-3eb60), suggesting that our cost functions are not effective for these cases. Our result opens up a new research question about what cost functions are more effective than simple cost functions (e.g., measuring output distance) we used in this work.

> In the majority of the cases we studied, patches are found faster when our cost functions are used than when costs are ignored.

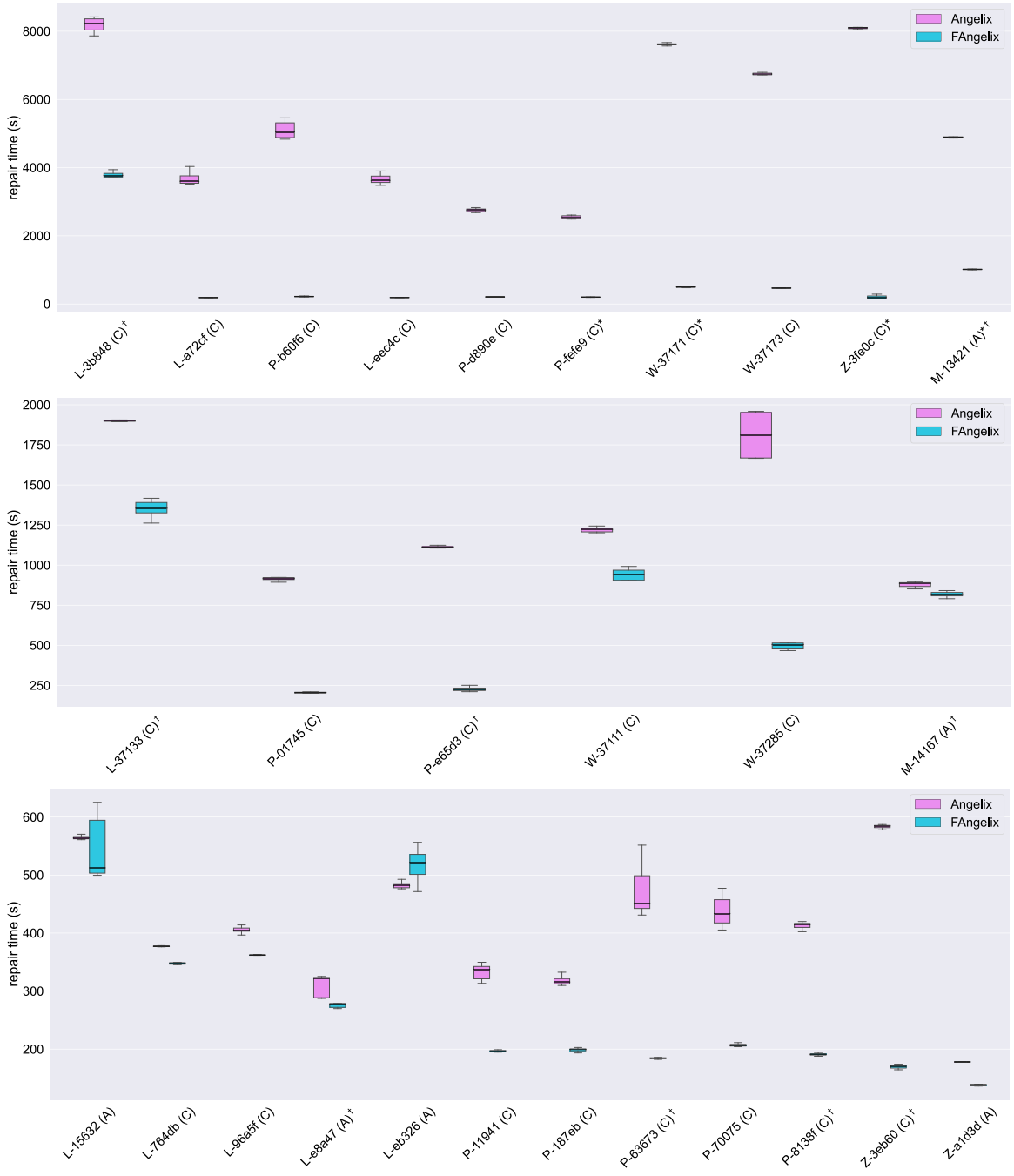



**Fig. 9.** Comparison of running time between Angelix and FAngelix when a patch was successfully generated in FAngelix. Note that in the four versions marked with asterisks (M-13421, P-fefe9, W-37171, and Z-3fe0c), Angelix terminated before a timeout occurs but failed to generate a patch. Notice that the maximum y-axis values are different between the three plots. FAngelix successfully generated correct patches in the nine versions marked with †. In the x-axis, buggy versions are annotated with the defect classes of generated patches ('C' for conditionals and 'A' for assignments).

### 5.6. RQ6: Comparison with the G&V approach

Fig. 12 shows the running time of GenProg in comparison with FAngelix. GenProg were successfully run in 38 versions of our subjects (GenProg did not terminate normally for the remaining 12 versions in our experiments), and the figure shows the results of these 38 versions. In general, a longer running time is observed in GenProg in comparison with FAngelix for the majority of the versions. When compared with median values, GenProg performs more slowly than FAngelix in 28 out of 38 versions (which amounts to 74%).

> FAngelix, which combines guided search and the early-termination policy, tends to generate repairs faster than GenProg.

## 6. Threats to validity and limitations

**External validity.** Our results may not generalize to other programs not used in our experiments, although the ManyBugs benchmark is currently a de facto standard benchmark for C programs. Similarly, our comparison results may not generalize to other program repair tools, in particular G&V program repair tools. To mitigate this threat, we compare the repair time of FAngelix with that of GenProg [26], and we could observe similar patterns from GenProg—as compared to FAngelix, a longer running time is generally observed.

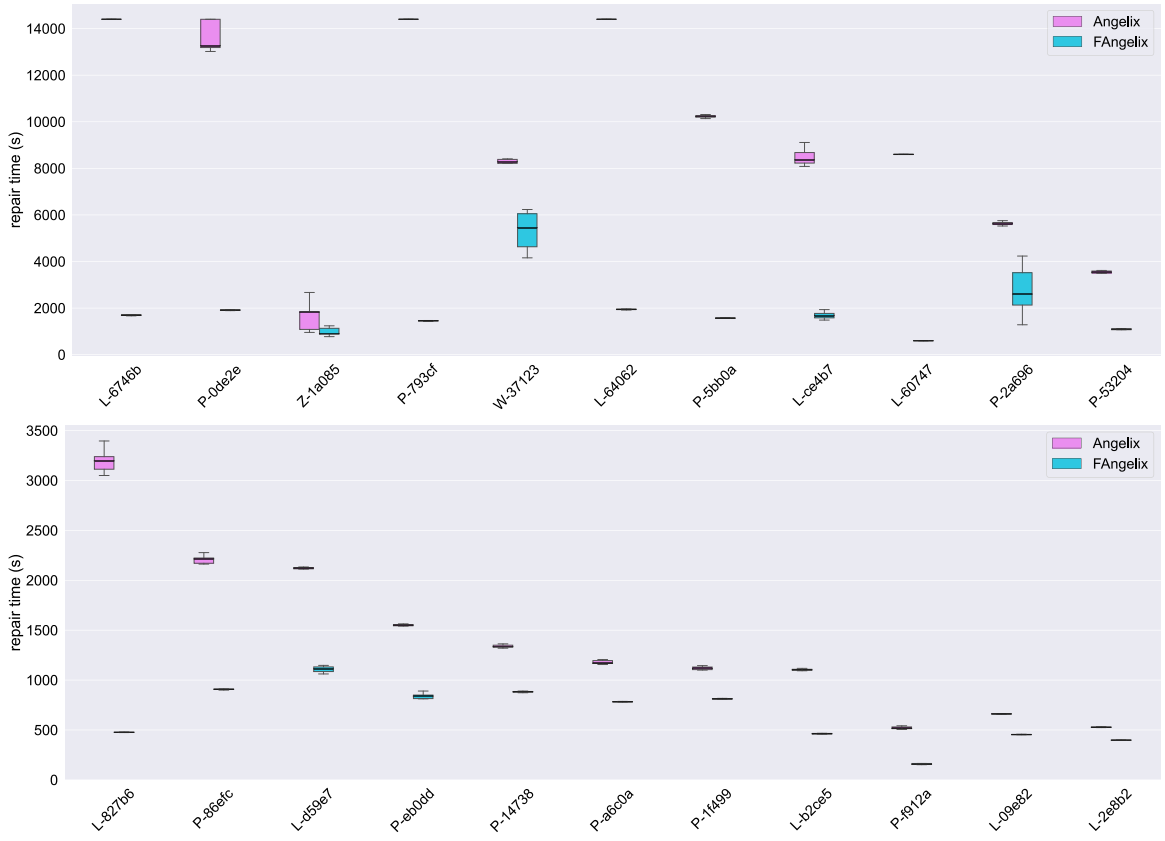



**Fig. 10.** Comparison of repair time between ANGELIX and FANGELIX for failed attempts of patch generation. Notice that the maximum y-axis values are different between the two plots.

Tool parameter values may also pose a threat to validity. Although we use the same parameter values for common modules (fault localization and patch synthesis) between FANGELIX and ANGELIX, the same parameter values may affect each tool differently. We leave it for future work to investigate this issue.

Our result may not generalize to other cost functions not used in our experiments. However, note that we chose cost functions that we think are the most straightforward. All our cost functions except for 2 GMP versions compute the distance between expected and actual output. Our study results suggest that such simple cost functions are generally effective in guiding a search for an angelic path. Nonetheless, finding an optimal cost function is not in the scope of this study, which we leave as future work.

**Internal validity.** There can be two possible explanations for the speedup of our approach; (1) our guided search algorithm and (2) the use of fast native execution instead of slow symbolic execution. In this study, we did not assess how each of these factors affects the efficiency of repair individually. However, we have pieces of evidence to believe that each factor has a positive effect on efficiency. Fig. 11 shows that out of six assignment bugs succeeded to be fixed by FANGELIX, three versions (L-15632, L-eb326, and L-e8a47) are fixed faster when the cost is used than when the cost is ignored. No visible difference is observed in the remaining three versions (M-13421, M-14167, and Z-a1d3d). Recall that for assignment bugs, FANGELIX uses symbolic execution. These results suggest a positive effect of our guided search. Meanwhile, FANGELIX fixes conditional bugs faster than ANGELIX in eight versions (L-96a5f, L-764db, P-70075, P-187eb, W-37173, W-37111, and W-37285), although the use of the cost does not improve efficiency in these versions. These results suggest a positive effect of native execution.

**Usability.** The fact that a user should provide a cost function can threaten the usability of our approach. However, this threat can be mitigated by writing test code in a stylized way; if expected and actual outputs are compared using a common subroutine (e.g., `equals(actual_out, expected_out)`), a cost can be obtained by replacing the subroutine with a cost function (e.g., `distance(actual_out, expected_out)` which returns the distance between `actual_out` and `expected_out`).

**Replicability.** To mitigate the threat to replicability, we provide our experimental scripts in the following URL: https://github.com/jyi/fangelix.

## 7. Discussion

In our experiments, we perform repair for each suspicious location one by one. However, such sequential search may not be optimal for runtime performance. Consider Fig. 13(a) where the two loop conditions are considered equally suspicious. In the sequential search, resources are wasted in investigating the first correct loop condition before investigating the second buggy loop condition. An alternative approach is to consider both expressions simultaneously. Our approach can easily support simultaneous search by extending the configuration to include multiple suspicious expressions (e.g., Fig. 13(b)) and having a group of multiple expressions investigated simultaneously.

Fig. 14(a) shows the result of simultaneous search for 9 LIBTIFF versions from which patches are generated by ANGELIX and FANGELIX.[9] The x-axis of the figure shows group sizes, where group size is defined as the number of suspicious locations that are investigated simultaneously. Meanwhile, the y-axis shows the average running time taken to generate a patch. It is observed that patches tend to be generated more quickly in both tools as more locations are investigated simultaneously.

[9] Similar patterns are also observed from other subjects.

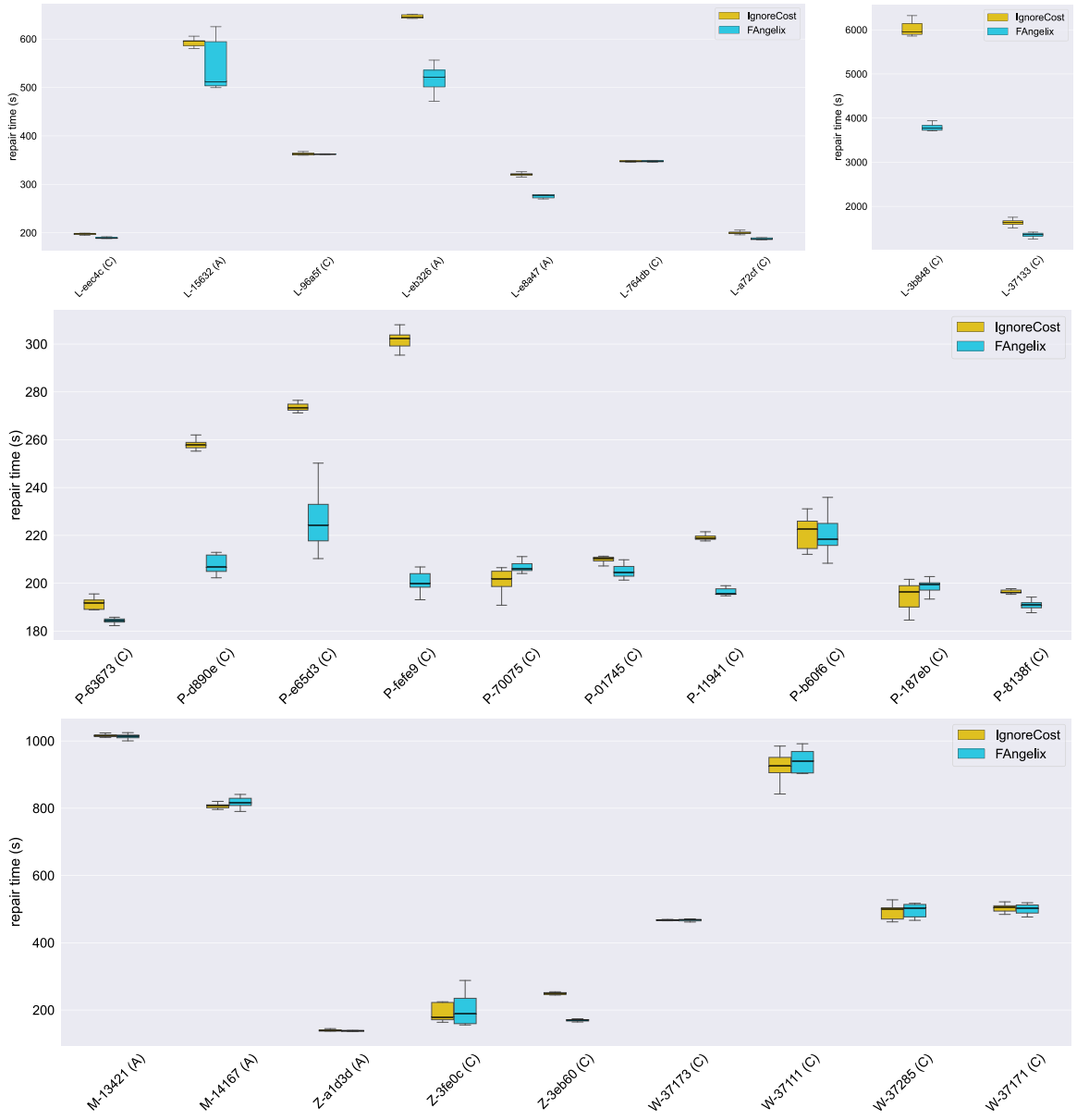



**Fig. 11.** Comparison of repair time between FAngelix and its variant where costs are ignored. Notice that the maximum y-axis values are different between the four plots.

However, there is a catch. Fig. 14(b) shows how the number of generated patches change as the group size increase. Our experimental result seems to suggest a tradeoff between effectiveness and repairability, while further investigation is left as future work.

## 8. Related work

### 8.1. Efficient patch space exploration

To efficiently explore the patch space, the following two general methods have been used, i.e., patch prioritization and patch space reduction. Note that most repair systems use both methods together. The idea of patch prioritization is to rank patch candidates according to their correctness likelihood. Prophet [28] is one of the earliest of such approaches, and ranks patch candidates based on a probabilistic model learned from existing patches. Similar approaches [4,32,33,37–40] that learn a ranked model from existing patches and other sources such as code comments (e.g., [33]) and Q&A sites (e.g., [38]) have been proposed. A more recent approach CapGen [32] builds an effective model by factoring in context information of buggy code and patch code into the model. Patch prioritization has also been widely used in constraint-based repair. Directfix [20] prioritizes a patch that makes minimal structural changes to the code, and similar approaches have been used in other tools using various forms of minimality [8–10].

In search-based approaches such as GenProg [26] and JAFF [41], patch prioritization is conducted via fitness functions. Each repair candidate is evaluated with a fitness function, and the neighborhood space of repair candidates with higher fitness values is explored with higher priority than the others in the subsequent repair process. GenProg uses a fitness function that returns the weighted sum of the number of passing and failing tests; different weights are used for passing tests and failing tests. JAFF uses a fitness function similar to the cost functions used in our experiments; the distance between expected and actual output is measured (JAFF also measures the size difference between the original and modified program). More recent works also consider other factors in their fitness functions, such as intermediate program states observed during test execution [42] and likely program invariants [43]. The cost functions used in our approach are similar to the fitness functions used in search-based approaches; both kinds of functions are used to explore the search space efficiently. The main novelty of our

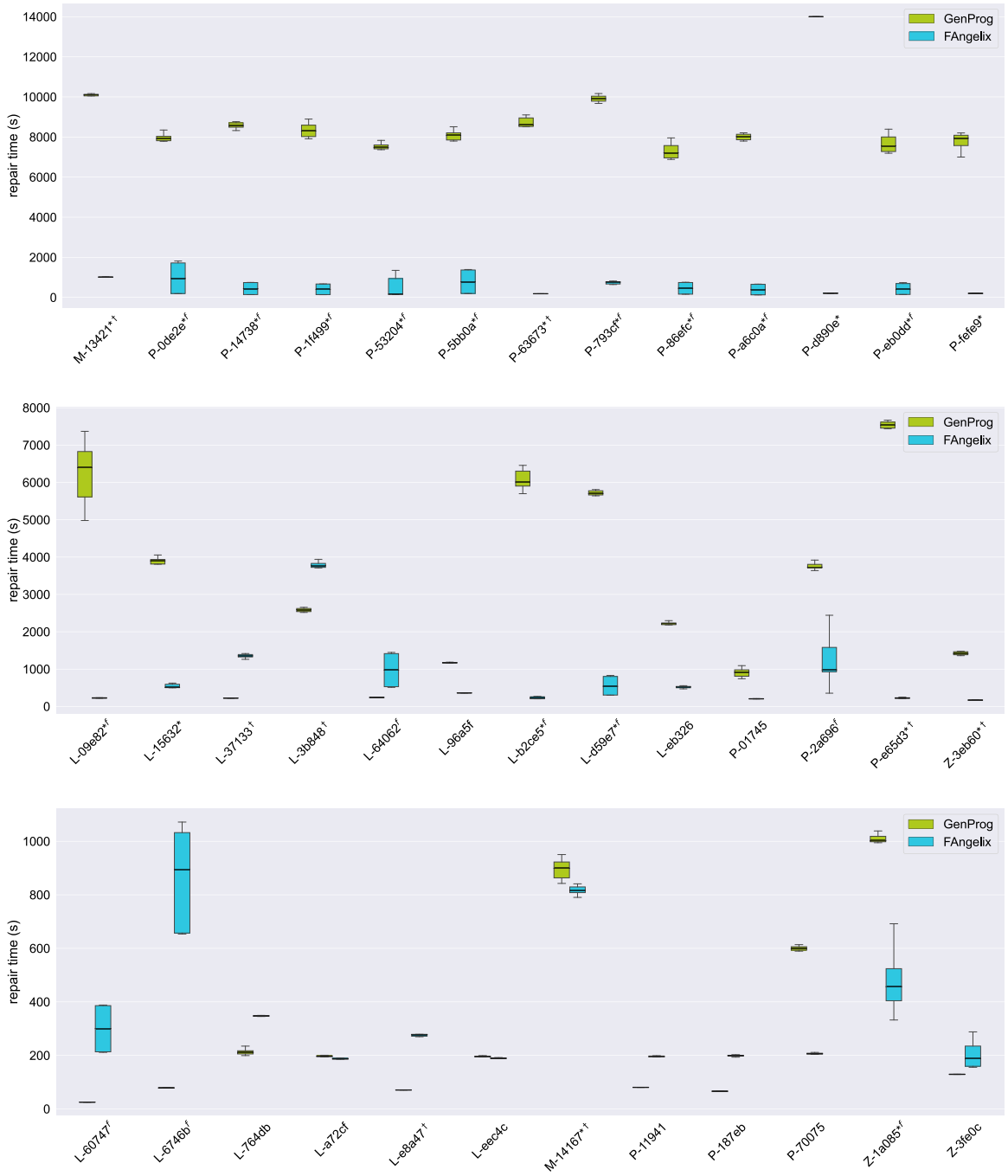



**Fig. 12.** Comparison of repair time between GenProg and FAngelix for 38 versions GenProg was successfully run. Notice that the maximum y-axis values are different between the three plots. The 21 versions failed to be repaired by GenProg are marked with $*$, and the 17 versions failed to be repaired by FAngelix are superscripted with $f$. The eight versions that are correctly fixed by FAngelix (see Section 5.2) are marked with †.

approach is in the use of cost functions in the context of constraint-based program repair, where we use a cost function to explore the specification space instead of the patch space.

To limit the patch space, most repair systems only use a *finite* number of repair patterns which are curated manually [44] or automatically [45]. In addition, Tan et al. [46] proposed not to use anti-patterns which often lead to overfitting patches. However, even a finite number of patterns easily induce huge patch space [29], and Genesis [45] addresses this issue by considering the size of the patch space involved with each repair pattern to discard those inducing large patch space. As a different approach, F1X [30] clusters patch expressions in the patch space into a smaller number of equivalent classes where the expressions in the same equivalent class exhibit the same behavior for a test under consideration.

Apart from repair patterns, a large number of suspicious locations can also enlarge the search space. While most APR systems use a fault localization technique and try to fix a location with a higher suspiciousness score than the others (spectrum-based fault localization techniques are typically used in the literature), suspiciousness scores are only approximate values and are not necessarily accurate. To mitigate this issue, [35] uses additional information (i.e., value ranges) to better rank suspicious locations; that is, a location $L$ is more suspected if a live variable at $L$ has a value $v$ outside the range $R$ where $v$ is obtained by running a failing test and $R$ is obtained by running passing tests.

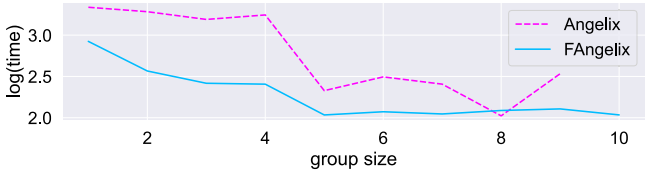
```
if (wrapper(td->td_nstrips == 1,
            /* Loc ID */ 574–13–574–31, ...)) { ... }
...
if (wrapper(td->td_nstrips > 1 && td->td_compresion == 1,
            /* Loc ID */ 589–13–590–31, ...)) { ... }
```

**(a) A buggy program with two suspicious expressions where the four numbers of the Loc ID describe the starting line number, the staring column number, the ending line number, and the ending column number**

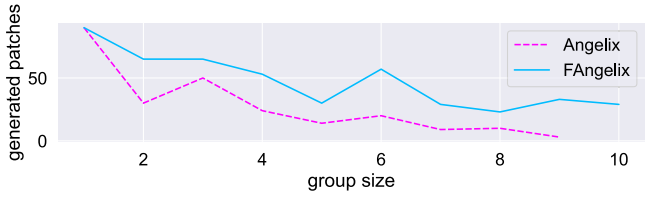
```
{"574–13–574–31": [1, 1, 0, 0], "589–13–590–31": [1, 1, 1, 0]}
```

**(b) An example of a configuration for Figure 13(a) containing the two suspicious locations (i.e., 574-13-574-31 and 589-13-590-31)**

**Fig. 13.** Simultaneous search for a patch specification.




**(a) Repair time**




**(b) The number of generated patches**

**Fig. 14.** Efficiency vs. effectiveness (the group size indicates the number of suspicious locations investigated simultaneously). In the top figure, log scale is used.

Constraint-based repair approaches like ours perform a staged search; patch space is explored only after a patch specification is obtained. Thus, exploring a patch space at a location $L$ is avoided if a patch specification is not found in $L$. For example, consider a suspicious statement at program location $L$, if $(E)$ {...}, where $E$ represents the expression used in the given buggy program. The patch space for $E$ is explored using program synthesis only after a patch specification for $E$ is obtained. Some G&V approaches [27,33,47] also perform a staged repair similarly. FAngelix reuses the fault localization and patch synthesis module of Angelix and hence explores the patch space in the same way as Angelix. However, FAngelix explores specification space differently from existing approaches for better efficiency.

### 8.2. Efficient specification space exploration

In constraint-based program repair, patch specification space is clearly separated out. To explore the patch specification space, the existing constraint-based repair approaches perform either bounded exhaustive search using symbolic execution [8,10,18] or explore a restricted search space as in Nopol [19]. In Nopol, given a suspicious conditional expression $e$, it is assumed that $e$ always returns the same boolean value at every occurrence of $e$. As a more advanced technique proposed by Mechtaev et al. [48], the use of second-order symbolic execution—where a symbolic variable can represent a patch candidate expression—reduces the patch specification space. This is because a symbolic execution path is not explored if there is no patch candidate expression in the patch space that can satisfy the path condition. Despite the reduction of patch specification space, their experimental results show that runtime is slower with second-order symbolic execution than with KLEE [6] unless a path bound (the maximum number of forks) is large (the critical bound is around 500). Meanwhile, some template-based repair approaches such as SPR [27] and ACS [33] also infer a patch specification for conditional expressions. These approaches, however, restrict the patch specification space by using predicate switching that flips only one bit at a time [49] or using similar heuristics. Furthermore, ACS considers only if conditions, ignoring loop conditions, unlike ours. Improving a patch synthesizer as done in ACS is orthogonal to our approach that finds a patch specification efficiently and effectively.

### 8.3. Test-suite reduction

Test-driven program repair systems repeat to run tests to validate patch candidates. Thus, test-suite reduction helps with reducing repair time as shown in [35]. Our approach is orthogonal to test-suite reduction, and can be used together with test-suite reduction.

## 9. Conclusion

In this paper, we have proposed a novel technique that expedites constraint-based program repair. Our experimental results show that our method, FAngelix, generates patches more efficiently, effectively, and accurately than Angelix, a state-of-the-art constraint-based program repair tool. The key enabler is our guided patch specification inference method. Using a search-based technique (i.e., MCMC sampling), we efficiently guide the search toward a patch specification. FAngelix also reports repair failure faster than Angelix, without sacrificing repairability. We conclude the paper with a couple of possible future work directions. We considered only first-order repairs in this work, and extending the approach to higher-order repairs is one direction we plan to pursue. Also, using other meta-heuristic search techniques other than MCMC sampling is another possible future work.

## CRediT authorship contribution statement

**Jooyong Yi:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing, Supervision, Project administration, Funding acquisition. **Elkhan Ismayilzada:** Data curation.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This work was partly supported by the National Research Foundation of Korea (NRF), South Korea grant funded by the Korea government (MSIT) (No. 2020R1F1A1066462, No. 2021R1A5A1021944, No. 2021R1A2C1009819) and Institute for Information & communications Technology Planning & Evaluation (IITP), South Korea grant funded by the Korea government (MSIT) (No. 2021-0-01001, Development of automatic software error repair technology that combines code analysis and error mining).

## References


[1] M. Harman, P. O'Hearn, From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis, in: SCAM (Source Code Analysis and Manipulation), IEEE, 2018, pp. 1–23.

[2] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, D. Rodriguez, Moving fast with software verification, in: NASA Formal Methods Symposium, Springer, 2015, pp. 3–11.

[3] K. Liu, S. Wang, A. Koyuncu, K. Kim, T.F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, Y.L. Traon, On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs, in: ICSE, 2020, pp. 615–627.

[4] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, X. Chen, Shaping program repair space with existing patches and similar code, in: ISSTA, ACM, 2018, pp. 298–309.

[5] K. Liu, A. Koyuncu, D. Kim, T.F. Bissyandé, TBar: revisiting template-based automated program repair, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 31–42.

[6] C. Cadar, D. Dunbar, D.R. Engler, KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: OSDI, 2008, pp. 209–224.

[7] C. Andrieu, N. De Freitas, A. Doucet, M.I. Jordan, An introduction to MCMC for machine learning, Mach. Learn. 50 (1–2) (2003) 5–43.

[8] S. Mechtaev, J. Yi, A. Roychoudhury, Angelix: scalable multiline program patch synthesis via symbolic analysis, in: ICSE, 2016, pp. 691–701.

[9] L. D'Antoni, R. Samanta, R. Singh, Qlose: Program repair with quantitative objectives, in: CAV, 2016, pp. 383–401.

[10] X.-B.D. Le, D.-H. Chu, D. Lo, C. Le Goues, W. Visser, S3: syntax-and semantic-guided repair synthesis via programming by examples, in: FSE, ACM, 2017, pp. 593–604.

[11] R. Abreu, P. Zoeteweij, A.J. Van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, IEEE, 2007, pp. 89–98.

[12] S. Jha, S. Gulwani, S.A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, in: ICSE, Vol. 1, IEEE, 2010, pp. 215–224.

[13] R. Alur, R. Bodík, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: Formal Methods in Computer-Aided Design, FMCAD, IEEE, 2013, pp. 1–8.

[14] W.K. Hastings, Monte Carlo sampling methods using Markov chains and their applications, Biometrika 57 (1) (1970).

[15] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, J. Chem. Phys. 21 (6) (1953) 1087–1092.

[16] W.R. Gilks, S. Richardson, D. Spiegelhalter, Markov Chain Monte Carlo in Practice, Chapman and Hall/CRC, 1995.

[17] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Trans. Softw. Eng. 28 (2) (2002) 183–200.

[18] H.D.T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, SemFix: program repair via semantic analysis, in: ICSE, 2013, pp. 772–781.

[19] J. Xuan, M. Martinez, F. Demarco, M. Clement, S.R.L. Marcote, T. Durieux, D.L. Berre, M. Monperrus, Nopol: Automatic repair of conditional statement bugs in java programs, IEEE Trans. Softw. Eng. 43 (1) (2017) 34–55.

[20] S. Mechtaev, J. Yi, A. Roychoudhury, DirectFix: Looking for simple program repairs, in: ICSE, 2015, pp. 448–458.

[21] A. Afzal, M. Motwani, K. Stolee, Y. Brun, C. Le Goues, SOSRepair: Expressive semantic search for real-world program repair, IEEE Trans. Softw. Eng. (2019).

[22] C.J.S. Leo Breiman, J. Friedman, R. Olshen, Classification and Regression Trees, Chapman and Hall/CRC, 1984.

[23] J.R. Quinlan, Simplifying decision trees, Int. J. Manmach. Stud. 27 (3) (1987) 221–234.

[24] C. Le Goues, N. Holtschulte, E.K. Smith, Y. Brun, P. Devanbu, S. Forrest, W. Weimer, The ManyBugs and IntroClass benchmarks for automated repair of C programs, IEEE Trans. Softw. Eng. 41 (12) (2015) 1236–1256.

[25] C.S. Timperley, S. Stepney, C.L. Goues, BugZoo: a platform for studying software bugs, in: ICSE, 2018, pp. 446–447.

[26] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: A generic method for automatic software repair, IEEE Trans. Softw. Eng. 38 (1) (2012) 54–72.

[27] F. Long, M. Rinard, Staged program repair with condition synthesis, in: ESEC/FSE, 2015, pp. 166–178.

[28] F. Long, M. Rinard, Automatic patch generation by learning correct code, in: POPL, 2016, pp. 298–312.

[29] F. Long, M.C. Rinard, An analysis of the search spaces for generate and validate patch generation systems, in: ICSE, 2016, pp. 702–713.

[30] S. Mechtaev, X. Gao, S.H. Tan, A. Roychoudhury, Test-equivalence analysis for automatic patch generation, TOSEM 27 (4) (2018) 15.

[31] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: Problem determination in large, dynamic internet services, in: DSN (Dependable Systems and Networks), 2002, pp. 595–604.

[32] M. Wen, J. Chen, R. Wu, D. Hao, S.-C. Cheung, Context-aware patch generation for better automated program repair, in: ICSE, ACM, 2018, pp. 1–11.

[33] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, L. Zhang, Precise condition synthesis for program repair, in: ICSE, 2017, pp. 416–426.

[34] L. Chen, Y. Pei, C.A. Furia, Contract-based program repair without the contracts, in: International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 637–647.

[35] B. Mehne, H. Yoshida, M.R. Prasad, K. Sen, D. Gopinath, S. Khurshid, Accelerating search-based program repair, in: ICST, IEEE, 2018, pp. 227–238.

[36] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Stat. 18 (1) (1947) 50–60.

[37] X.D. Le, D. Lo, C. Le Goues, History driven program repair, in: SANER (Software Analysis, Evolution and Reengineering), 2016, pp. 213–224.

[38] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, H. Mei, Fixing recurring crash bugs via analyzing Q&A sites, in: ASE, IEEE, 2015, pp. 307–318.

[39] Y. Pei, C. Furia, M. Nordio, Y. Wei, B. Meyer, A. Zeller, Automated fixing of programs with contracts, IEEE Trans. Softw. Eng. 40 (5) (2014) 427–449.

[40] M. White, M. Tufano, M. Martinez, M. Monperrus, D. Poshyvanyk, Sorting and transforming program repair ingredients via deep learning code similarities, in: SANER (Software Analysis, Evolution and Reengineering), IEEE, 2019, pp. 479–490.

[41] A. Arcuri, Evolutionary repair of faulty software, Appl. Soft Comput. 11 (4) (2011) 3494–3514.

[42] E.F. de Souza, C.L. Goues, C.G. Camilo-Junior, A novel fitness function for automated program repair based on source code checkpoints, in: Genetic and Evolutionary Computation Conference, 2018, pp. 1443–1450.

[43] Z.Y. Ding, Y. Lyu, C. Timperley, C. Le Goues, Leveraging program invariants to promote population diversity in search-based automatic program repair, in: IEEE/ACM International Workshop on Genetic Improvement (GI), IEEE, 2019, pp. 2–9.

[44] D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in: ICSE, 2013, pp. 802–811.

[45] F. Long, P. Amidon, M. Rinard, Automatic inference of code transforms for patch generation, in: FSE, ACM, 2017, pp. 727–739.

[46] S.H. Tan, H. Yoshida, M.R. Prasad, A. Roychoudhury, Anti-patterns in search-based program repair, in: FSE, 2016, pp. 727–738.

[47] J. Hua, M. Zhang, K. Wang, S. Khurshid, Towards practical program repair with on-demand candidate generation, in: ICSE, ACM, 2018, pp. 12–23.

[48] S. Mechtaev, A. Griggio, A. Cimatti, A. Roychoudhury, Symbolic execution with existential second-order constraints, in: ESEC/FSE, ACM, 2018, pp. 389–399.

[49] X. Zhang, N. Gupta, R. Gupta, Locating faults through automated predicate switching, in: ICSE, ACM, 2006, pp. 272–281.