Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis

Sergey Mechtaev Jooyong Yi Abhik Roychoudhury
School of Computing, National University of Singapore, Singapore
{mechtaev,jooyong,abhik}@comp.nus.edu.sg

ABSTRACT

Since debugging is a time-consuming activity, automated program repair tools such as GenProg have garnered interest. A recent study revealed that the majority of GenProg repairs avoid bugs simply by deleting functionality. We found that SPR, a state-of-the-art repair tool proposed in 2015, still deletes functionality in their many "plausible" repairs. Unlike generate-and-validate systems such as GenProg and SPR, semantic analysis based repair techniques *synthesize* a repair based on semantic information of the program. While such semantics-based repair methods show promise in terms of quality of generated repairs, their scalability has been a concern so far. In this paper, we present Angelix, a novel semantics-based repair method that scales up to programs of similar size as are handled by search-based repair tools such as GenProg and SPR. This shows that Angelix is more scalable than previously proposed semantics based repair methods such as SemFix and DirectFix. Furthermore, our repair method can repair multiple buggy locations that are dependent on each other. Such repairs are hard to achieve using SPR and GenProg. In our experiments, Angelix generated repairs from large-scale real-world software such as wireshark and php, and these generated repairs include multi-location repairs. We also report our experience in automatically repairing the well-known Heartbleed vulnerability.

1 Introduction

The once-futuristic idea of automated program repair is gradually becoming a reality. Various automated repair tools, such as GenProg [14], PAR [21], *relifix* [39], SemFix [26], Nopol [8], DirectFix [24] and SPR [23], to name only a few, have been introduced recently. These automated repair methods can be classified into the following two broad methodologies, i.e., search-based methodology (e.g., GenProg, PAR, and SPR) and semantics-based methodology (e.g., SemFix, Nopol, and DirectFix). Search-based repair methodology (also known as generate-and-validate methodology) searches within a search space to *generate* a repair candidate and *validate* this repair candidate against the provided test-suite. Meanwhile, the semantics-based repair methodology *synthesizes* a repair using semantic information (via symbolic execution and constraint solving).

Classifying repair methods into search based repair and semantics based repair is somewhat analogous to classification of software testing into search-based testing and symbolic-execution-based testing [28]. While such a classification may be a bit coarse, it helps us understand the current trends in automated program repair. In [14], GenProg, a prominent search-based repair tool, is shown to be scale to large-scale real-world software such as `php` and `wireshark`. Meanwhile, SemFix [26], the first semantics-based repair tool, is shown to be more efficient than GenProg in terms of repairability (which is higher as more buggy programs can be repaired) and running time, although SemFix was applied only to relatively small programs. Afterwards, in both methodologies, the importance of high-quality repairs began to be considered, resulting in another search-based repair tool, PAR [21], and semantics-based repair tool, DirectFix [24]. Finally, a research effort [25] argued for explicitly defining *defect classes* (*i.e.*, which defects will be fixed by a given repair method) while constructing repair methods.

Currently, research in automated program repair considers all the three attributes – scalability (should scale to large real-world programs), repairability (should repair a large number of defects possibly by covering many defect classes), and the quality of repairs (should produce repairs which make less changes to the program, delete less functionality, and are more likely to be accepted by developers). Note that some of these attributes can be deemed to be somewhat qualitative in nature — nevertheless, it is very important to consider them while building a new repair method. As an example, the latest search-based repair tool, SPR [23], generates more repairs (as compared to GenProg), and the generated repairs are more frequently functionally-equivalent to developer-provided repairs (rather than merely passing the provided tests), when applied to large-scale real-world software. Meanwhile, in the case of the semantics-based methodology, low scalability has been the main source of criticism, despite its promising results in terms of high repairability (*e.g.*, SemFix [26]) and the high quality of repairs (*e.g.*, [24] which repairs a buggy program by making provably minimal changes to the program).

We show in this paper how the semantics-based repair methodology can also scale up to the same level as the most advanced search-based repair tools such as SPR and Gen-

Prog. Semantics based repair methods often work by extracting a *repair constraint* typically via symbolic execution. This repair constraint acts as a specification to guide program synthesis - so a patch satisfying the repair constraint can be synthesized. The key enabler for scalable multiline bug fix in this paper, is our novel lightweight repair constraint that we call an *angelic forest*. This angelic forest is automatically extracted via symbolic execution. As compared to the repair constraints used in the previous work [24, 26], the angelic forest is simpler, and its size is *independent* of the size of the program under repair, thereby making our repair method scale. Our angelic forest, despite its simplicity, contains enough semantic information to enable multi-location bug fix. Among existing search-based repair tools, SPR does not support multi-line fixes. While GenProg [14] can change multiple locations of the program, a recent study on GenProg repairs [33] shows that seemingly complex repairs generated from GenProg are in the overwhelming majority of cases in fact functionally equivalent to single line modification.

When evaluated with the largest of the GenProg ICSE2012 subjects, our open-source repair tool Angelix successfully generated repairs including on wireshark and php subjects. The number of repairs generated by Angelix (28) is larger than in GenProg (11), and also generally comparable to SPR (31). While in one subject (libtiff), Angelix generated more repairs than SPR, and in another subject (php), SPR generated more repairs. In the remaining 3 subjects, both tools produced the same number of repairs.

More importantly, we note that even though a recent work [33] points to functionality deleting repairs by GenProg, the SPR tool [23] (which was produced by the same authors) itself was found to generate many functionality deleting repairs, because it generates many trivial branch conditions. Such trivial branch conditions (conditions which are always true or always false) introduce functionality deletion; *e.g.*, consider the following SPR repair for a libtiff defect where the shaded part is the fix inserted by SPR.

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1]
    && !(1))
```

We found that in SPR the overall functionality-deleting repair rate across the GenProg benchmark subjects is 45%. In fact, in the `libtiff` subject, the percentage of functionality deleting repairs in the SPR tool [23] goes up to an alarming 80% !! In contrast, our semantic analysis based repair tool produced functionality-deleting repairs significantly less frequently when the same tests were used (21%). For the aforementioned defect, Angelix synthesizes a patch that is identical with the developer-provided patch (shown in Figure 3b), which does not delete functionality. Furthermore, the repairs generated by Angelix include five multi-location bugs which have not been fixed by the existing tools. Last but not the least, we report that the well-known Heartbleed vulnerability [1] was automatically fixed by our tool, generating a repair that is smiliar to the developer-provided patch. To the best of our knowledge, ours is the first work that reports the automated repair on Heartbleed. Overall, we present a semantic analysis based program repair method which balances the requirements of scalability (repairing large programs), repairability (repairing a large number of defects) and patch quality (changing the functionality of the program in a way developers would agree with, instead of simply deleting functionality). Our tool Angelix and its experimental data are available at the following web site: `www.comp.nus.edu.sg/~abhik/tools/angelix/`

2 Motivating Example

Figure 1 shows code changes made to fix coreutils bug 13627. In the buggy version, the call of `xzalloc` (line 4), which allocates a block of memory, causes a segmentation fault. A fix involves adding an if conditional before the problematic call to `xzalloc` (line 5). Only when variable `max_range_endpoint` has a non-zero value, `xzalloc` can be called in the fixed version. In addition to adding an if conditional, the fix requires removing an existing if statement (lines 1–2). Without this removal, `max_range_endpoint` is overwritten with a non-zero value of `eol_range_start` (line 2), and as a result, the new if conditional "`if (max_range_endpoint)`" cannot successfully prevent the problematic call to `xzalloc`.

This simple example demonstrates the complexity of multiline repairs fixing multiple buggy locations. The key difficulty is that a change made in one location can also change the remaining program execution that should proceed to be repaired. More conceptually speaking, the fix space of a given buggy program keeps changing along with the program change made at each buggy location. The state-of-the-art search-based repair algorithm such as SPR [23] (also known as the generate-and-validate methodology) is currently restricted to fixing a single location. It is unclear, as stated in [23], how a search-based repair algorithm can be extended to fix multiple-location bugs such as the one shown in Figure 1a, while maintaining its efficiency. Meanwhile, among the state of the art of semantics-based repair methodology such as SemFix [26] and DirectFix [24], DirectFix already supports multiple-location fix. Essentially, DirectFix maintains all semantic information of the program (in the form of a logical formula), and this makes it possible to keep track of how the fix space changes. Thus SemFix is more scalable and applies one line fixes, while DirectFix is less scalable but can produce multi-line fixes.

In this paper, we discuss how semantics-based repair can scale, while preserving its ability to repair multiple locations. Figures 1b–1d show at a high level how our repair algorithm generates a repair from our running example. The generated repair shown in Figure 1d is functionally equivalent to the developer-provided repair, despite its cosmetic differences. The first piece of a repair, "`if (0)`" in line 1, makes the statement in line 2 skipped over, which is functionally equivalent to removing the corresponding statement. The second piece of a repair, "`if (!(max_range_endpoint == 0))`", is also functionally equivalent to the developer-provided repair, `if (max_range_endpoint)`. Here, the cosmetic difference is merely due to our current implementation of the component-based synthesis algorithm we use to synthesize a repair.

Our repair algorithm starts from transforming the original buggy program into a functionally equivalent one shown in Figure 1b where we add an if conditional, `if (1)`, before each unguarded assignment statement (this is heuristics we currently use). Afterwards, our repair algorithm replaces user-configured n most suspicious expressions—chosen based on the result of statistical fault localization—with symbolic variables, as shown in Figure 1c where conditional expressions and the right-hand side of an assignment are replaced with symbolic variables. The user of our repair algorithm can con-

```
1  − if (max_range_endpoint < eol_range_start)
2  −   max_range_endpoint = eol_range_start;
3
4  − printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
5  + if (max_range_endpoint)
6  +   printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(a) The developer-provided bug patch for coreutils bug 13627 where multiple locations are repaired

```
1  if (max_range_endpoint < eol_range_start)
2     max_range_endpoint = eol_range_start;
3
4  if (1)
5     printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(b) The buggy version after semantics-preserving transformation (the shaded part is added)

```
1  if (α)
2     max_range_endpoint = β;
3
4  if (γ)
5     printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(c) Suspicious expressions are replaced with symbolic variables

```
1  if (0)
2     max_range_endpoint = eol_range_start;
3
4  if (! (max_range_endpoint == 0))
5     printable_field = xzalloc(max_range_endpoint/CHAR_BIT+1);
```

(d) A repair generated from our repair algorithm; expressions in the shaded areas are synthesized from our repair tool, Angelix.

Figure 1: Motivating example

figure the number and kinds of suspicious expressions that can be made *symbolic*; such expressions include conditional expressions, right-hand sides of assignments, and function parameters. Our repair algorithm proceeds to run symbolic execution over the program in Figure 1c with provided tests to collect the semantic information necessary to repair the given buggy program. Using this extracted semantic information, we synthesize repair expressions. To synthesize a repair, we use component-based patch synthesis algorithm based on MaxSMT as in our prior work [24]. This results in a repair close to the original program, because the structures of the original buggy expressions are maximally preserved. The resultant small patches can bring in various benefits such as improved maintainability of patches (simple patches are easier to understand than complex patches), and reduced risk for regression (simple patches are less likely to change the correct behavior than complex patches).

2.1 Concise Semantic Signature for Repair

In order to synthesize a repair, our repair algorithm collects the following pieces of semantic information of the program. First, we need to know whether for each test, there exists a program path through which a given test passes. Our repair algorithm detects such test-passing paths via controlled symbolic execution—"controlled" in the sense that we control which execution paths are explored during symbolic execution by installing symbolic variables (in our example, α, β, and γ). In our running example, given a program of Figure 1c, symbolic execution explores different paths at the if conditionals in line 1 (if (α)) and line 4 (if (γ)). If a test-passing path is not detected, we make the next (user-configured) n suspicious expressions as symbolic, and repeat the procedure to find test-passing paths. On the other side, the existence of a test-passing path π that goes through the installed symbols implies the existence of a concrete value for each symbol that makes the test pass. As the second piece of semantic information, we infer these values (called *angelic values*) using a constraint solver. Lastly, we need to know the program state (called *angelic state*) at each installed symbol in the test-passing path. For example, in order to synthesize a repair expression, !(max_range_endpoint == 0), at line 4 of Figure 1d, the value of the variable max_range_endpoint should be known. Our repair algorithm collects the values of the visible program variables at each symbol-installed program location. These variables are used as synthesis ingredients when synthesizing repair expressions. The following shows the semantic signature of our running example when two tests (t_1 and t_2) are provided.

$$\begin{aligned} t_1 : \{&\pi_1 : \langle(\alpha, False, \sigma_1), (\gamma, False, \sigma_2)\rangle,\\ &\pi_2 : \langle(\alpha, True, \sigma_3), (\beta, 0, \sigma_4), (\gamma, False, \sigma_5)\rangle\}\\ t_2 : \{&\pi_3 : \langle(\alpha, False, \sigma_6), (\gamma, True, \sigma_7)\rangle,\\ &\pi_4 : \langle(\alpha, True, \sigma_8), (\beta, 3, \sigma_9), (\gamma, True, \sigma_{10})\rangle\},\end{aligned}$$

where t_i referes to a test, π_i denotes a test-passing path, and $\sigma_i : Variables \to Values$ denotes an angelic state.

The preceding semantic signature—which we call an angelic forest as defined in Definition 3—concisely captures all three pieces of semantic information we need to synthesize a repair. First, the fact that there exist two execution paths (π_1 and π_2) that make test t_1 pass is encoded in $t_1 : \{\pi_1, \pi_2\}$. Similarly, test t_2 can also pass in two execution paths, π_3 and π_4. Note that the suggested repair shown in Figure 1d follows path π_1 in test t_1, and π_3 in t_2. Second, the concrete value of each symbol is denoted at each test-passing path. For example, in path π_1, symbol α and γ should have value False, as denoted with $\pi_1 : \langle(\alpha, False), (\gamma, False)\rangle$. The concrete value of symbol β does not appear because statement max_range_endpoint = β of Figure 1c is not executed in path π_1. Meanwhile, in path π_2, the values of all three symbols appear as denoted with $\pi_2 : \langle(\alpha, True), (\beta, 0), (\gamma, False)\rangle$. Lastly, angelic state σ_i informs about the values of variables to use in repair synthesis. The same variable can have different values along a path, and that is why each instance of a symbol is associated with its own angelic state. In our coreutils example, σ_2(max_range_endpoint) is zero, and this is why the suggested repair expression, !(max_range_endpoint == 0), returns the concrete value of γ, False, as specified in π_1.

2.2 Reasons for Scalability

As will be shown in the experimental results (Section 5), our repair method can handle programs as large as wireshark (2814 KLoC), while it generates multi-location repairs. There are multiple reasons why our repair method scales. First, we use a *lightweight semantic signature* for program synthesis. Compare our semantic signature with the one used in DirectFix [24] which can also synthesize multi-location repairs. The semantic signature used in DirectFix is essentially the semantics of the whole program. There, the relationship between each and every expression appearing in the program is maintained, unlike in our new semantic signature. As a result, the semantic signature of DirectFix becomes more lengthy and complex, as the size of the program increases. It it important to note that the semantic signature is the specification for repair synthesis in the sense that a synthesized repair should respect the provided semantic signature, as explained with our running example. Our lightweight semantic signature reduces the burden of the repair synthesizer, resulting in more efficient repair synthesis.

Second, our repair algorithm performs *controlled symbolic execution* with a few selected suspicious expressions, instead of usual symbolic input. Using this controlled symbolic execution, we explore only a restricted number of feasible execution paths involving only the selected few suspicious expressions. Also, we initially perform symbolic execution only with a subset of the provided test-suite to reduce the running time of symbolic execution. Only when some of remaining tests fail with the synthesized repair, we perform additional symbolic execution with these failing tests.

Lastly, our repair algorithm initiates repair synthesis only when there exists an angelic forest—the semantic signature for repair. The absence of an angelic forest for a chosen n suspicious locations implies that it is not possible to repair the bug by changing these n locations. Symbolic execution finds an angelic forest (or proves the absence of an angelic forest) efficiently by exploring only feasible execution paths. Our repair algorithm does not waste the resources to synthesize a repair if there is no angelic forest.

We note that each of these afore-listed techniques is the improvement or extension of earlier work by us and others. As already mentioned, our novel lightweight program-size-independent semantic signature is the improvement of the heavyweight semantic signature used in our prior work DirectFix [24]. We also mention that the controlled symbolic execution was first introduced in our prior work, SemFix [26], although there a symbol is installed only at one location, and as a result, multi-location repair was not possible. Lastly, our repair strategy to ignore repair-wise infeasible suspicious locations has a similarity with Nopol [8] and SPR [23]. While detailed comparison will be provided in Section 8, Nopol and SPR currently cannot fix multi-location bugs. Furthermore, multi-location fix seems fundamentally difficult in Nopol and SPR, due to their weaker semantic signatures that do not capture the dependence between multiple program locations. The unique combination of our novel semantic signature with the existing techniques enables scalable multi-location bug fixing.

3 Background

We use techniques and tools for Partial Maximum Satisfiability Modulo Theories for our repair algorithm. First, *Satisfiability Modulo Theories (SMT)* is a problem of finding a satisfying assignment of a given a logical formula with respect to the provided background theories. *Partial MaxSMT (pMaxSMT)* for two sets of SMT clauses (*soft* and *hard*) is a problem of finding an assignment of the variables that satisfy all hard clauses and maximum possible number of the soft clauses.

To synthesize a patch, we use component-based repair synthesis algorithm (CBRS) [24] which is a generalization of component-based program synthesis [17] to program repair. In CBRS, a *component* is a variable, a constant or a term over components and operations defined in the given background theory; e.g., 1, x, $(*_1 + *_2)$, and $(\mathit{if}\ *_1\ \mathit{then}\ *_2\ \mathit{else}\ *_3)$ where $*_i$ refers to the input to a component. An expression is formed by connecting multiple components. For example, the diagram in Figure 2 shows that expression $x > y$ is formed by connecting the following three components: $(*_1 > *_2)$, x, and y. Each component c has one output c^{out} and one or more inputs c_i^{in}. We denote the number of inputs of a component c with $NI(c)$. To represent the connection between components, the input and output of components are associated with distinct variables called *location variables*. For example, input c_i^{in} is associated with its location variable lc_i^{in}, and output c^{out} with lc^{out}. Two components are considered connected if and only if the location variable of one component has the same value as the location variable of another component.

To make sure that a synthesized expression is well-formed, CBRS imposes a well-formedness constraint (ϕ_{wpf}) in which C denotes the set of all available components:

$$
\begin{aligned}
\phi_{wpf} &\overset{def}{=} \phi_{range} \wedge \phi_{cons} \wedge \phi_{acyc} \\
\phi_{range} &\overset{def}{=} \bigwedge_{c \in \{v\} \cup C} \Big(0 \leq lc^{out} < |C| \ \ \wedge \bigwedge_{k \in [1, NI(c)]} 0 \leq lc_k^{in} < |C| \Big) \\
\phi_{cons} &\overset{def}{=} \bigwedge_{(c,s) \in C \times C, c \neq s} lc^{out} \neq ls^{out} \\
\phi_{acyc} &\overset{def}{=} \bigwedge_{c \in C, k \in [1, NI(c)]} lc^{out} > lc_k^{in}
\end{aligned}
$$

where the range constraint (ϕ_{range}) places all components inputs and outputs within a legal range, the consistency constraint (ϕ_{cons}) ensures that the output of each component has a distinct location, and the acyclicity constraint (ϕ_{acyc}) prohibits cyclic connections. CBRS also imposes the semantics constraint for each component, and the connections constraint (ϕ_{conn}) that connects location variables with their corresponding components.

$$
\phi_{conn} \overset{def}{=} \bigwedge_{\substack{(c,s) \in C \times \{v\} \cup C \\ k \in [1..NI(s)]}} lc^{out} = ls_k^{in} \Rightarrow c^{out} = s_k^{in}
$$

Lastly, structural constraints capture the structure of the original buggy program. For example, the structural constaints for the expression $x + y$ are the following:

$$
\phi_{struct} \overset{def}{=} l{+}_1^{in} = lx^{out} \wedge l{+}_2^{in} = ly^{out}
$$

To synthesize a patch for a buggy program using CBRS, the structural constraints are passed to a Partial MaxSMT solver as soft constraints and the rest of the constraints are passed as hard constraints. Then, the solver finds a new program that satisfies the synthesis specification and is syntactically closest to the original buggy program.

4 Methodology

Our repair methodology consists of the following 4 steps: (1) program transformation, (2) fault localization, (3) extracting a repair constraint, and (4) patch synthesis. In the first step, we perform semantics-preserving program transformation to expand the defect class our repair algorithm can fix. For example, we showed in Section 2 that "if (1)" can be added before each unguarded statement. More generally, our repair framework is transparent to the addition of more semantics-preserving program transformation schemes. In the second step, we perform statistical fault localization. We use the Jaccard formula [7], considered most effective for automated program repair according to [32]. Since our repair algorithm modifies buggy expressions, we apply the Jaccard formula at the expression level, instead of at the statement level. The last two steps distinguish semantics-based repair methods from search-based repair methods such as GenProg and SPR. Semantics-based methods extract a repair constraint from the program under repair typically via symbolic execution. This repair constraint acts as a specification to guide program synthesis—so a patch satisfying the repair constraint can be synthesized.

The key novelty of our repair method is our new lightweight repair constraint that we call an *angelic forest.* The size of this angelic forest is *independent* of the size of the program under repair. This is the main reason why our new repair method can scale. Our angelic forest, despite its simplicity, contains enough semantic information to enable multi-location bug fix. In the following, we formally define our angelic forest (Definition 3) based on the definition of an angelic value (Definition 1) and an angelic path (Definition 2).

Definition 1 (Angelic Value [6]). *Let P be a program, t be a failing test case, e be a program expression and e^k be its k-th appearance in the execution trace of t. Angelic value α is such that replacing expressions e^k with α during the execution of t in the trace makes P pass test t.*

Definition 2 (Angelic Path). *Let E be a set of program expressions of program P, and t be a test case for P. An angelic path $\pi(t, E)$ is a set of triples (e^k, v, σ) where e^k is the k-th instance of an expression $e \in E$ appearing in the execution trace of test t, v is an angelic value of e^k, and function, $\sigma : Variables \rightarrow Values$, represents angelic state at e^k which is a mapping from visible variables at the location of e^k to their values. For these triples (e^k, v, σ) in an angelic path $\pi(t, E)$, the following property holds: replacing all e^k in an angelic path with their corresponding angelic values v makes (1) program P passes the test t and (2) visible variables x at the location of e^k have values $\sigma(x)$.*

Definition 3 (Angelic Forest). *Let E be a set of program expressions of program P, and t be a test case for P. Angelic forest A_t for test t is a collection of angelic paths $\{\pi_1(t, E), \ldots, \pi_n(t, E)\}$.*

4.1 Angelic Forest Extraction

We extract an angelic forest via *controlled* custom symbolic execution – "controlled" in the sense that instead of initiating symbolic execution with symbolic input, we install symbols at a few suspicious program locations—chosen based on a statistical fault localization result—to control the execution paths to be explored during symbolic execution. Algorithm 1 shows how we extract an angelic forest. Performing controlled symbolic execution produces a pair of a path condition *pc* and an actual output O_a of the program (line 3). Given the expected output O_e available in the test, we find a model of $pc \wedge O_a = O_e$ (i.e., model M in line 6) via a constraint solver. This model is used to extract an angelic path, and thereafter grow the angelic forest (line 7). Recall that an angelic forest is a set of angelic paths, each of which is a set of triples consisting of an instance of a suspicious expression e^k, its angelic value (the value e^k should return to pass the test), and angelic state σ at e^k, that is, a mapping from visible variables at the location of e^k to their values.

Algorithm 1 Angelic forest generation

```
Input: program $P$, test case $(I, O_e)$
Input: a set of suspicious expressions $E$
Output: angelic forest $A$
 1: while there is an unexplored path $\wedge \neg$timeout do
 2:     // perform controlled symbolic execution
 3:     $(pc, O_a) \leftarrow$ ControlledSymExe$(I, E)$
 4:     $R \leftarrow pc \wedge O_a = O_e$
 5:     if $R$ is satisfiable then
 6:         $M \leftarrow$ GetModel$(R)$ // via a constraint solver
 7:         $A \leftarrow A \cup$ ExtractAngelicPath$(M)$
 8:     end if
 9: end while
10: return $A$
```

Algorithm 2 Our custom symbolic execution

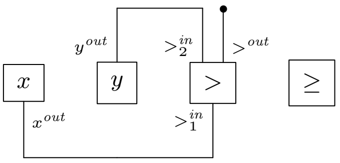
```
Input: $e^k$ is a $k$-th instance of expression $e$
Input: a set of suspicious expressions $E$
Input: $\sigma_{sym}$ : $Variables \rightarrow ConcreteValues \cup SymbolicValues$
Output: the concrete/symbolic value of expression $e^k$
 1: function EvaluateExpr($e^k$, $E$, $\sigma_{sym}$)
 2:     if $e^k \in E$ then // if $e^k$ is suspicious
 3:         for $x \in$ visible variables at the location of $e^k$ do
 4:             AddToPathCondition($x[\![e^k]\!] == \sigma_{sym}(x)$)
 5:         end for
 6:         // install a symbol for $e^k$
 7:         return NewSymbolicVariable($e^k$)
 8:     else// if $e^k$ is not suspicious
 9:         // Evaluate $e^k$ as usual
10:         return EvaluateExprConventionally($e^k$, $\sigma_{sym}$)
11:     end if
12: end function
```

Since the conventional symbolic execution can neither install symbols for chosen suspicious expressions nor maintain the angelic states of suspicious expressions, we extend the conventional symbolic execution, as shown in Algorithm 2. In our custom symbolic execution, symbols are installed during symbolic execution by replacing the value of each instance of a suspicious expression with a fresh symbol (line 7). If a given e^k is not a suspicious expression, our custom symbolic execution evaluates e^k in the same way as in the conventional symbolic execution (line 10). In addition, we maintain the angelic states of suspicious expressions by augmenting the path condition (line 4). For each visible variable x at the location of e^k (the k-th instance of suspicious expression e), we extend the path condition with $x[\![e^k]\!] == \sigma_{sym}(x)$, where $x[\![e^k]\!]$ represents the variable x in the context of e^k, and $\sigma_{sym}(x)$ the concrete/symbolic value of x evaluated during symbolic execution. Solving the resultant augmented path conditions via a constraint solver produces an angelic forest. We implement our custom symbolic execution on top of KLEE [4].

4.2 Patch Synthesis

Once an angelic forest is obtained, we feed it to our repair synthesizer as a synthesis specification. More specifically, a synthesized repair, when executed, follows one of the angelic paths for each test, thereby all tests pass. In these angelic paths, each repaired expression returns its corresponding angelic value specified in the corresponding angelic path.

Our repair synthesizer is an implementation of component-based repair synthesis (CBRS) described in Section 3. CBRS



Input		Output
x	y	
2	1	True
2	2	False
1	2	False

Figure 2: The circuit diagram on the left represents expression $x > y$ which satisfies the specification shown on the right as a table. Unconnected components are ignored.

Table 1: The example of an angelic forest

Path	k	$x[\![e^k]\!]$	$y[\![e^k]\!]$	Angelic
1	1	2	1	True
	2	2	2	False
2	1	1	2	False

views a program as a circuit of primitive components such as variables and operators. For example, the circuit diagram of Figure 2 shows the circuit for a program expression $x > y$. The boxes and lines represent components and connections, respectively. The goal of CBRS, given an original buggy program, components, and a specification of the program to be synthesized, is to search for connections between components that (1) satisfy the given specification and (2) minimally differ from the connections of the original buggy program. As an example, consider a buggy expression $x \geq y$. The table of Figure 2 shows the desired specification for the expression, and CBRS modifies the connection of the variables x and y from the component '$\geq$' to the component '$>$', because $x > y$ satisfies the given specification.

The specification of CBRS is provided in the form of an angelic forest extracted by Algorithm 1. Table 1 shows an example of an angelic forest. For simplicity, only one expression e is considered suspicious in this example (the extension to multiple expressions is straightforward). The first two columns of the table show a path ID and the instance ID of e, respectively. In this example, Path 1 executes expression e twice (that is, k is either 1 or 2), while Path 2 executes e only once. The angelic forest includes two visible program variables, x and y, as the program state at the suspicious location. The values of these variables are shown in the $x[\![e^k]\!]$ and $y[\![e^k]\!]$ columns. Notation $x[\![e^k]\!]$ represents the variable x in the context of e^k, the k-th instance of e. The last column, Angelic, shows the angelic value of e^k.

CBRS performs the search for connections using a constraint solver. Given our example angelic forest, we generate the following constraint:

$$\Big(x[\![e^1]\!] = 2 \land y[\![e^1]\!] = 1 \land e^1 = \mathit{True} \land$$
$$x[\![e^2]\!] = 2 \land y[\![e^2]\!] = 2 \land e^2 = \mathit{False}\Big)$$
$$\lor \Big(x[\![e^1]\!] = 1 \land y[\![e^1]\!] = 2 \land e^1 = \mathit{False}\Big)$$

More generally, we generate the following constraint, given an angelic forest $\{\pi_i(t, E)\}$ (recall that each angelic path $\pi_i(t, E)$ in an angelic forest is a set of triples (e^k, v, σ)):

$$\bigvee_{\pi_i(t,E)} \bigwedge_{(e^k,v,\sigma)} \Bigg(\Big(\bigwedge_{x \in dom(\sigma)} x[\![e^k]\!] = \sigma(x)\Big) \land e^k = v\Bigg),$$

where v is the angelic value of e^k, and $dom(\sigma)$ refers to the domain of σ, the mapping from the visible variables at the location of e^k to their values.

There can be multiple patches satisfying a given repair constraint. In such cases, CBRS finds a patch requiring minimal changes by using MaxSMT solver. The ability to maximally preserve the original source code is important for two reasons. First, our hypothesis is that such a minimal patch would be preferred by developers. Minimal patches are easier to validate and they are less likely to change the correct behavior of the original program than more complex patches as demonstrated in [24]. Second, when synthesizing a repair for multiple suspicious expression, MaxSMT-based repair serves as fault localization, that is the repair algorithm simultaneously identifies which expressions to modify and how to modify them. Without this property, synthesizer would always modify all the suspicious expressions making milti-location repair not practical due to the complexity of patches. As will be shown in Section 5, this way of synthesis provides higher-quality repairs than SPR.

4.3 Optimization

To control the number of symbolic execution sessions, we use the following iterative approach. First, we start from a small subset of the test suite that provides the highest coverage of the suspicious locations. Then, we infer angelic forest for this reduced test suite and synthesize a patch. If the generated patch causes a regression in the whole test suite, we add the counter-example test to the test suite. We repeat these steps until all test cases become passing. Regarding running time, there is one more advantage of semantics-based methods. Contrasting to search-based methods where the software under repair is rebuilt and retested frequently due to a high number of repair trails, our semantics-based method finds a repair in one or a small number of trials, and the cost for rebuilding and retesting is significantly smaller.

Because of type coercion and the absence of a separate boolean type in C programming language, it is difficult to distinguish between types of program expressions. On the other side, knowing precise types increases the probability of synthesizing correct repair as well improves the synthesis performance. For this reason, we analyze the usage of suspicious expression and visible variables to collect type constraints. Then, these type constraints are used to infer more precise types for program expressions and variables. As an example, we assign a boolean type to the expressions used as if conditions.

4.4 Soundness and Completeness

While the size of an angelic forest independent of the size of the program, it also under-approximates the fix space—that is, it cannot capture whole (possibly infinite) set of values for the suspicious expressions that make the test pass. Our repair method based on an angelic forest is sound in the sense that the repair obtained by our repair method indeed passes all the provided tests. However, our repair method is incomplete in the sense that it may not produce some repairs, due to the under-approximation of angelic values used in an angelic forest, that can otherwise be synthesized.

5 Experimental Results

We evaluate our repair method to answer the following two research questions.

RQ1. Can our repair method generate repairs from large-scale real-world software?

RQ2. Can our repair method fix multi-location bugs?

5.1 Experimental Subjects

The first 4 columns of Table 2 show our subject programs, the size of each program in LoC, and the number of tests and buggy versions of each subject (in the Tests and Versions columns, respectively). Our subjects are taken from the GenProg ICSE2012 benchmark [13]. These subjects have been also used in the literature to evaluate other repair tools such as GenProg [14] and SPR [23]. In particular, wireshark and php are among the largest subjects in the benchmark. We use these large subjects to evaluate the scalability of our repair method. We omit three subjects of the benchmark (python, lighttpd, and fbc) because we could not run these subjects on KLEE [4]. KLEE currently cannot support all library functions. Note that this limitation of KLEE is orthogonal to our repair approach.

We use the same subjects to evaluate our second research question, i.e., multi-location repairability. Furthermore, in addition to these subjects in the GenProg benchmark, we add 3 multi-location bugs extracted from CoREBench [3] to our subject list. The reason we added multi-location bugs additionally is that the GenProg benchmark does not have many multi-location bugs in the fix space of our tool. We describe the defect class of our repair tool in Section 5.4.

5.2 Tests and the Correctness of Patches

The "Tests" column of Table 2 shows the number of tests of each subject in the GenProg benchmark. We rectified the original test scripts delivered in the GenProg benchmark to address the problems pointed out in [33] such as the weak proxy problem. Meanwhile, each coreutils version available in CoREBench contains a failing test that can reproduce the defect. We use these failing tests and the existing tests available in the subject. All the repairs generated from our tool are manually inspected for its correctness. We consider a repair correct only if the generated patch is functionally equivalent to the developer-provided patch.

5.3 Experimental Configurations

Our repair tool allows to control the following parameters of our repair algorithm — the maximum number of suspicious locations that can be repaired at the same time, the kinds of suspicious expressions, and the kinds of (semantics-preserving) program transformation.

First, for the maximum number of suspicious locations, we used the value between 1 and 10 (inclusive). Second, for the kinds of suspicious expressions, we used the following three levels. A higher level is more inclusive. At the lowest level, we allow only conditional expressions to be considered suspicious. At the next level, we also consider the right-hand side expressions of assignment statements. At the highest level, we also consider function parameters. At all levels, only side-effect/function-call free expressions are considered. Lastly, for the semantics-preserving program transformation, we allow to add "if (1)" before each unguarded statement. We also allow to add "if (0) break;" at the end of a loop body to be able to produce a repair requiring to break a loop. We provided our tool with the names of buggy source code files (which are known through developer-provided fixes), as in the previous studies [14, 23, 40]. All our experiments were performed on Intel Xeon E5-2660 2.20GHz CPU with Ubuntu 14.04 64-bit operating system. We used 12 hours as the timeout of each repair session.

5.4 Defect Class

As pointed out in [25], defining defect classes supported by a repair algorithm helps evaluate the efficacy of a repair algorithm (how effectively bugs in the target defect class can be repaired), and compare multiple repair algorithms one another (which repair algorithm generates repairs more effectively for the target defect class). The defect class of our repair algorithm can easily be defined in terms of the fix that can be synthesized. Our repair tool can synthesize side-effect/function-call free expressions that can be composed of boolean/arithmetic/relational operators, variables available, and constants. Also, by using semantics-preserving program transformation (i.e., adding if (1) before unguarded statements), fixes requiring statement deletion is effectively included in our defect class, as shown with the motivating example in Section 2. However, our repair tool currently cannot add a new statement/variable. The "W/I Our Defect Class" column of Table 2 shows the number of defects of each subject that are in our defect class. We manually inspected each developer-provided fix to check whether the corresponding defect is in our defect class or not. Although there can be other possible fixes different from a developer-provided fix, it is infeasible to consider all unknown possible fixes. Thus, we additionally only inspected fixes from other repair tools (SPR, GenProg, and AE [40]) and ours. The number of defects within our defect class is less than the number of buggy versions (shown in the "Versions" column), because some bug fixes in the benchmark require adding new statements/variables.

5.5 Results from the GenProg Benchmark

Table 2 shows our results from the GenProg benchmark. The first five columns are already explained earlier, and self-explanatory. We only mention that subjects of the table are sorted by their sizes. The "Fixed Defects" column shows the number of fixed defects by our tool, Angelix, and other tools—SPR, GenProg, and AE. Similarly, the "Equiv. to Developer Fixes" column shows the number of fixes functionally-equivalent to the developer-provided fixes out of the fixed defects. The results from other tools (SPR, GenProg and AE) are taken from [23]. Lastly, "Time" column shows the average running time of our tool for each subject, when repairs were found. The running time of the other tools are available in their respective papers [14, 23, 40], although each tool is experimented on a different type of machine.

In all subjects with different sizes between 77 KLoC and 28214 KLoC, our tool successfully generated repairs for some defects. Our tool generated repairs for most defects in our defect class (28 out of 32), and more than third of these repairs (10 out of 28) are functionally equivalent to developer-provided repairs. Three defects in our defect class were not fixed due to imprecise statistical fault localization (e.g., buggy initialization of a global variable was not ranked high). One remaining defect requires modifying a string value (a character sequence) in a way that cannot be handled by our current solver (the length of the string should change in a fix). As shown in the Time column, the average running time of our repair tool is about half an hour, when a repair is found.

Table 2: Experimental results

Subject	LoC	Tests	Versions	W/I Our Defect Class	Fixed Defects				Equiv. to Developer Fixes				Time (min)
					Angelix	SPR	GenProg	AE	Angelix	SPR	GenProg	AE	
wireshark	2814K	63	7	4	4	4	1	4	0	0	0	0	23
php	1046K	8471	44	12	10	18	5	7	4	8	1	2	62
gzip	491K	12	5	2	2	2	1	2	1	1	0	0	4
gmp	145K	146	2	2	2	2	1	1	2	1	0	0	14
libtiff	77K	78	24	12	10	5	3	5	3	1	0	0	14
Overall			82	32	28	31	11	19	10	11	1	2	32

Table 3: The number of defects exclusively repaired by each repair tool across the subjects

Subject	Angelix	SPR	GenProg	AE
wireshark	0	0	0	0
php	0	4	0	0
gzip	1	0	0	0
gmp	0	0	0	0
libtiff	5	0	0	0
Overall	6	4	0	0

> Angelix, an implementation of our new semantics-based repair algorithm, successfully generates repairs from 5 real-world software as large as 77–28214 KLoC in 32 minutes on average. This result shows that a semantics-based repair can scale.

Angelix fixed 2 multi-location bugs of the GenProg benchmark. We show these results along with the results from the multi-location bugs of coreutils in Section 5.6.

5.5.1 Comparison with Other Repair Tools

Repairability. When compared with the state-of-the-art repair tool, SPR, our tool shows higher repairability (more defects are repaired in our tool) in libtiff (10 vs 5), and lower repairability in php (10 vs 18). In the remaining 3 subjects, both tools shows the same repairability. This varying repairability across the subjects is related to the different defect classes of Angelix and SPR. For example, the defect class of SPR contains inserting a function call such as memset, and 5 php defects are included in this defect class. Meanwhile, Angelix can fix multiple buggy locations, and two libtiff multi-location defects are exclusively fixed by our tool. More generally, Table 3 shows the number of defects exclusively repaired by each repair tool across the subjects. Our tool produced the most number of unique repairs, as compared to SPR, GenProg and AE.

Repair Quality. We also qualitatively compare the repairs from Angelix and SPR. Figure 3 shows a buggy location of libtiff-d13be72c-ccadf48a in (a), the repair generated by our tool in (b), and the repair generated by SPR in (c). The difference between the original code and each repair is shaded. The SPR repair looks problematic, because it simply deletes functionality by disabling the block of code in the then branch. Indeed, this patch is not functionally equivalent to the developer-provided patch. Still, such an overfitting repair [38] (an incorrect repair that merely passes the provided tests) can be helpful in debugging, because the user can at least see that the (incorrectly) repaired if conditional may be buggy. However, compare this SPR repair to the repair generated by our tool shown in Figure 3b. Our repair spots the buggy location more precisely down to "td->td_nstrips > 1". This is because our repair tool generates a repair that is close to the original buggy expression by using a MaxSMT solver. As a result, a problematic buggy location can be pinned down more precisely. In fact, our repair is identical with the developer-provided repair in this case.

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
```

(a) The buggy location of libtiff-d13be72c-ccadf48a

```
if (td->td_nstrips > 2
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
```

(b) The repair generated by our tool, Angelix

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1]
    && !(1))
```

(c) The repair generated by SPR

Figure 3: Comparison between repairs from Angelix and SPR

Table 4: The number of functionality-deleting repairs

Subject	Angelix			SPR		
	Fixes	Del	Per	Fixes	Del	Per
wireshark	4	1	25%	4	1	25%
php	10	3	30%	18	7	39%
gzip	2	0	0%	2	1	50%
gmp	2	0	0%	2	0	0%
libtiff	10	2	20%	5	4	80%
Overall	28	6	21%	31	13	42%

Incorrect repairs that merely delete functionality are common in SPR repairs. Table 4 compares the number of repairs

Table 5: Experimental results for multi-location defects.

Defect	Fixed Expressions
libtiff-4a24508-cc79c2b	2
libtiff-829d8c4-036d7bb	2
coreutils-00743a1f-ec48bead	3
coreutils-1dd8a331-d461bfd2	2
coreutils-c5ccf29b-a04ddb8d	3

that delete functionality between our tool and SPR. In each tool (the "Angelix" and "SPR" column, respectively), we list the number of fixes generated in each tool (the Fixes column), the number of functionality-deleting fixes (the Del column), and the percentage of functionality-deleting fixes out of generated fixes (the Per column). In five subjects used in our experiments, 42% of SPR-generated repairs delete functionality, and in the libtiff subject, the percentage goes up to 80%. Even if the three omitted subjects (python, lighttpd, and fbc) are also considered, the percentage of functionality deleting repairs stays even at a high rate of 45%. GenProg and AE also often generate functionality-deleting repairs, as reported in [33]. In comparison, Angelix generates functionality-deleting repairs less frequently (21%).

Angelix is not only scalable but also less frequently generates functionality-deleting repairs than the existing tools such as SPR and GenProg.

5.6 Results from Multi-Location Bugs

Table 5 shows the experimental results for multi-location defects of the GenProg benchmark and coreutils. The "Fixed Expressions" column shows the number of expressions fixed by our tool. Angelix produced a repair functionally equivalent to the developer-provided one for coreutils-00743a1f-ec48bead. Meanwhile, in coreutils-1dd8a331-d461bfd2, while two conditional expressions are repaired in a functionally similar way to the developer patches, the output message is not corrected in our repair, because this message is not part of the the oracle in the tests used for repair. In coreutils-c5ccf29b-a04ddb8d, the developer-provided repair uses function calls that are not used in our repair.

We note that the number of defects covered by our multi-location defect class is limited at least in the two benchmarks we investigated (the GenProg benchmark and CoREBench). Many developer-provided fixes for multi-line defects involve adding new variables, statements, and functions. We believe that the research in automatic patching should be developed into such more sophisticated patches, and our multi-location defect class is on the pathway toward such a direction. To the best of our knowledge, only our repair tool can currently generate (non-functionality-deleting) fixes for multi-location bugs in large-scale real-world software.

6 Experience with the Heartbleed Bug

We applied our repair tool to a buggy version of OpenSSL (OpenSSL-1.0.1-beta1) that has the infamous Heartbleed bug. Heartbleed is considered one of the most dangerous in the annals of security vulnerabilities, because attackers can exploit Heartbleed to steal important confidential data, including login cookies, passwords, and private cryptographic keys, without leaving a trace from numerous servers depending on OpenSSL to run their services. We report that we could automatically fix the Heartbleed bug using our repair tool. To the best of our knowledge, this is the first work that reports the automated repair on Heartbleed.

```
1  if (hbtype == TLS1_HB_REQUEST) {
2     ...
3     memcpy (bp, pl, payload);
4     ...
5  }
```

(a) The buggy part of the Heartbleed-vulnerable OpenSSL

```
if (hbtype == TLS1_HB_REQUEST
   && payload + 18 < s->s3->rrec.length) {
   /* receiver side: replies with TLS1_HB_RESPONSE */
}
```

(b) A fix generated by our tool, Angelix

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;
...
if (hbtype == TLS1_HB_REQUEST) {
   /* receiver side: replies with TLS1_HB_RESPONSE */
}
else if (hbtype == TLS1_HB_RESPONSE) {
   /* sender side */
}
return 0;
```

(c) The developer-provided repair

Figure 4: The Heartbleed bug and their fixes

The Heartbleed bug is an instance of a buffer over-read (CWE-126), one of common weakness of C/C++ programs. Exploiting this weakness, attackers can read beyond the region of a buffer that is originally intended by the programmers. Figure 4a shows where this weakness exists in the Heartbleed-vulnerable OpenSSL. The main culprit is memcpy(bp, pl, payload) (line 3) where attackers can assign payload (the third parameter of memcpy that sets the number of bytes to copy from the target memory region) a larger value than the size of buffer pl, the target memory region. The programmer of OpenSSL made a (common) mistake of not putting a bounds check before this problematic memcpy.

We applied Angelix to OpenSSL for repairing the Heartbleed bug. We obtained tests from [2], and added four more tests to cover missing corner cases. Figure 4b shows the fix generated by our tool in the shaded area. With this fix, memcpy cannot be invoked if payload is larger than is allowed by the TLS/DTLS network protocol (the buggy code of OpenSSL is the implementation of these protocols). Our repair synthesizer could compose this repair with payload and s->s3->rrec.length, both of which are in the scope at the fixed if conditional (they appear in the other parts of the buggy function). In comparison, the developer-provided repair is shown in the shaded area of Figure 4c. In both repairs, the failure of the bounds check, which is performed by the added conditional, makes the receiver simply return zero, instead of replying with a response packet. Based on our experience with Heartbleed, we make the following assessment.

Automated repair techniques, such as Angelix, are powerful enough to fix some of well-known and serious software vulnerabilities like Heartbleed.

7 Threats to Validity

Since we used for our experiments the subject programs in the existing benchmark previously used to evaluate GenProg, AE, and SPR, the validity of our experimental results are limited in the same way as for the results of the other tools obtained using the same benchmark. That is, our results may not generalize to other subjects, although our repair tool successfully generated repairs for a small number of defects of Coreutils and OpenSSL. However, we note that the GenProg benchmark we used for our experiments is one of the most extensive one available in the literature (the ManyBugs benchmark [15] has also been released very recently as an extension of the GenProg benchmark). Meanwhile, our results can be affected by the configuration of our tool (e.g., the maximum number of suspicious locations). However, given the pervasiveness of cloud computing environments such as Amazon EC2, this threat related to tool configuration does not seem as severe as traditionally believed, since our tool can be run in parallel in the cloud, with each node being assigned a different tool configuration.

8 Related Work

GenProg [14] performs search based repair through genetic programming algorithm. It is the first general-purpose program repair tool that showed the defects of large-scale real-world software can be automatically fixed. Subsequently, RSRepair [31] and AE [40] replace the genetic programming algorithm of GenProg with random search and adaptive repair search strategies, respectively. While these repair methods scale well, a recent study [33] revealed that the quality of the repairs generated from these tools are quite poor—the majority of these repairs simply delete functionality. While PAR [21], another search-based repair tool, uses human patch templates to improve the repair quality, and validates its improved repair quality through a user study. However, as argued by [25], the results of the user study on PAR in fact only shows that PAR repairs more resemble human patches than GenProg repairs (because repairs are generated through human patch templates). The latest search-based repair tool, SPR [23], more often generates repairs that are functionally equivalent to developer-provided repairs than its preceding search-based repair tools, by taking into account the (partial) semantics of conditionals, that is, the branches that should be taken to pass the tests. However, we found that SPR still generates many functionality-deleting repairs, because it often generates trivial branch conditions such as if (... && !(1)). As reported earlier, about half of the reported SPR repairs (45%) delete functionality.

Meanwhile, semantics based repair methodology has shown its promise in its high quality of repairs. The first approach towards semantics based program repair was SemFix [26] where a combination of symbolic analysis and constraint solving was proposed to produce one line fixes. DirectFix [24] used a MaxSMT solver to synthesize minimal changes to the program which make the program pass all tests. DirectFix removed the one-line-fix restriction of SemFix, and yet gave an approach which is substantially less scalable than SemFix. Our new repair tool addresses this scalability problem, while retaining the ability to produce multi-line fixes. While the key enabler for scalability is our novel repair constraint representation (i.e, angelic forest), we also capitalize on the techniques successfully used in our previous work SemFix [26], such as controlled symbolic execution. This makes the Angelix repair tool very scalable, while generating high quality multi-line repairs.

Nopol [8] uses an angelic value to synthesize a repair, similar to Angelix. However, the expressiveness of our angelic forest is substantially larger than the one of an angelic value. Consequently, Angelix can repair more bugs than Nopol. For example, Nopol cannot repair if-conditionals whose fix should take different directions at different times during execution, because a single angelic value is not expressive enough to capture such a repair requirement. Meanwhile, SPR [23] maintains a sequence of angelic values of a conditional expression, instead of a single angelic value. As a result, SPR can handle a broader class of defects than Nopol. However, it cannot fix multi-location bugs whose fix often requires information about dependence between multiple suspicious locations, which cannot be captured only with sequences of angelic values. Even for single-location bugs, SPR often generates functionality-deleting repairs. Furthermore, even when functionality is not deleted, SPR often generates templated repairs such as if (... || regex_len == 42), which may work only for a specific test, and break for fresh input not covered by the existing tests. On the contrary, our MaxSMT-based repair synthesizer often synthesizes a repair close to the original buggy expression.

Apart from general-purpose repair tools like ours, there are also other repair approaches targeted for specific types of defects (e.g., buffer overflow) or specific application domains (e.g., web applications) [5, 9, 11, 18, 27, 30, 35–37]. Also, many previous works assume the existence of formal specification or contracts [10, 12, 16, 19, 22, 29, 34] unlike test-driven approaches such as ours. Lastly, MintHint [20] suggests a repair hint instead of a patch by allowing some tests to remain failing and thereby performing statistical analysis for a class of semi-repairs satisfying this relaxed requirement.

9 Conclusion

In this paper, we have described how a semantics-based repair method can scale to large-scale real-world software. The key enabler for this scalability is our novel lightweight repair constraint called 'angelic forest'. We have shown through experiments that our repair method successfully generate repairs from various real-world software, including wireshark and php, which are the largest programs to which automated repair tools have been applied. Furthermore, on top of providing scalability, our repair method also produces higher quality repairs than the existing scalable repair tools such as SPR and GenProg; as compared to these existing tools, our repair tool produced functionality-deleting repairs less frequently in our experiments. We also have shown that our repair tool successfully fixed multi-location bugs in real-world software, which was not possible in the existing repair tools. Last but not least, we have reported the successful patching of the well-known Heartbleed bug, using our repair tool.

ACKNOWLEDGEMENTS

We thank Shin Hwei Tan, Dipanjan Das and G Sri Shaila for assisting in experiments. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

[1] The heartbleed bug. http://heartbleed.com, 2014.

[2] M. Bland. https://code.google.com/p/mike-bland/source/browse/heartbleed/, 2015.

[3] M. Böhme and A. Roychoudhury. Corebench: Studying complexity of regression errors. In *ISSTA*, pages 105–115, 2014.

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

[5] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with Jolt. In *ECOOP*, pages 609–633, 2011.

[6] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *ICSE*, pages 121–130, 2011.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, pages 595–604, 2002.

[8] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA*, pages 30–39, 2014.

[9] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, pages 233–244, 2006.

[10] B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *ICSE*, pages 855–858, 2008.

[11] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *ASE*, pages 307–318, 2015.

[12] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188, 2011.

[13] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. GenProg ICSE2012 benchmark. http://dijkstra.cs.virginia.edu/genprog/resources/genprog-icse2012-benchmarks/, 2012.

[14] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, pages 3–13, 2012.

[15] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE TSE*, 2015.

[16] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *FASE*, pages 267–280, 2004.

[17] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

[18] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400, 2011.

[19] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[20] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *ICSE*, pages 266–276, 2014.

[21] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811, 2013.

[22] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *FMCAD*, pages 91–100, 2011.

[23] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC-FSE*, 2015.

[24] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE*, 2015.

[25] M. Monperrus. A critical review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the problem statement and the evaluation of automatic software repair. In *ICSE*, pages 234–242, 2014.

[26] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *ICSE*, pages 772–781. IEEE Press, 2013.

[27] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, 2008.

[28] A. Orso and G. Rothermel. Software testing: A research travelogue (2000–2014). In *FOSE*, pages 117–132, 2014.

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

[30] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102, 2009.

[31] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *ICSE*, pages 254–265, 2014.

[32] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *ISSTA*, pages 191–201, 2013.

[33] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, pages 24–36. ACM, 2015.

[34] H. Samimi, E. D. Aung, and T. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[35] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE*, pages 277–287, 2012.

[36] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[37] A. Smirnov and T. cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS*, 2005.

[38] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *FSE*, 2015.

[39] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In *ICSE*, 2015.

[40] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*, pages 356–366, 2013.