MintHint: Automated Synthesis of Repair Hints

Shalini Kaleeswaran Varun Tulsian* Aditya Kanade
Indian Institute of Science, India
{shalinik,varuntulsian,kanade}@csa.iisc.ernet.in

Alessandro Orso
Georgia Institute of Technology, USA
orso@cc.gatech.edu

ABSTRACT

Being able to automatically repair programs is at the same time a very compelling vision and an extremely challenging task. In this paper, we present MintHint, a novel technique for program repair that is a departure from most of today's approaches. Instead of trying to fully automate program repair, which is often an unachievable goal, MintHint performs statistical correlation analysis to identify expressions that are likely to occur in the repaired code and generates, using pattern-matching based synthesis, *repair hints* from these expressions. Intuitively, these hints suggest how to rectify a faulty statement and help developers find a complete, actual repair.

We also present an empirical evaluation of MintHint in two parts. The first part is a user study that shows that, when debugging, developers' productivity improved manyfold with the use of repair hints—instead of traditional fault localization information alone. The second part consists of applying MintHint to several faults in Unix utilities to further assess the effectiveness of the approach. Our results show that MintHint performs well even in common situations where (1) the repair space searched does not contain the exact repair, and (2) the operational specification obtained from the test cases for repair is incomplete or even imprecise, which can be challenging for approaches aiming at fully automated repair.

Categories and Subject Descriptors:
D.2.5 [Software Engineering]: Testing and Debugging

General Terms:
Debugging aids, Diagnostics, Symbolic execution

Keywords:
Program repair, statistical correlations, program synthesis, repair hints

1 INTRODUCTION

Debugging is an expensive activity that can be responsible for a significant part of the cost of software maintenance [39]. It is therefore not surprising that researchers and practitioners alike have invested a great deal of effort in developing techniques that can improve the efficiency and effectiveness of debugging (*e.g.,* [3, 6, 7, 11, 15, 25, 29, 31, 32, 34, 35, 42]). In particular, in recent years, there has been a growing interest in automated program repair techniques (*e.g.,* [6, 11, 29, 31, 32, 35]). Although these techniques have been shown to be effective, they suffer from one or more of the following limitations. First, some techniques rely on the existence of a specification for the program being debugged, which is rarely the case in practice. Second, techniques that do not rely on specifications tend to "overfit" the repair to the set of existing test cases, which is likely to affect the general validity of the repair. Third, because they are looking for a complete repair, most existing techniques must perform a search over a repair space that is large enough to include the (unknown) repair. For non-trivial repairs, this can make the technique either ineffective (if the bound on the repair space used by the tool is too small) or too expensive to be used in practice (if the bound on the repair space used/required is too large).

```
65  char esc(char *s, int *i) {
66      char result;
67      if (s[*i] != ESCAPE)
68          result = s[*i];
69      else
70          if (s[*i] == ENDSTR) // Localization rank 3
71              result = ESCAPE;
72          else {
73              *i = *i + 1; // Localization rank 4
74              if (s[*i] == 'n')
75                  result = NEWLINE;
76              else
77                  if (s[*i] == 't') // Localization rank 2
78                      result = TAB;
79                  else
80                      result = s[*i]; // Localization rank 1
81              }
82      return result;
83  }
```

Figure 1: Function esc from program replace (version 23), which contains a faulty statement at line 70. The ones shown are the line numbers of function esc in the actual code.

To address these limitations of existing techniques, in this paper we propose MintHint, a novel, semi-automated approach to program repair. MintHint is a departure from most of today's program repair techniques as it does *not* try to find a complete repair, which we have observed to be an unachievable goal in many, if not most, cases due to technical and practical reasons. Instead, MintHint aims to *generate repair hints that suggest how to rectify a faulty statement and help developers find a complete, actual repair*.

As an example, consider function esc from a faulty version of program replace [19], shown in Figure 1. Function esc takes as input a string and an index into the string and checks whether the character at the index is a special character (*e.g.,* a newline). If so, it returns a program-specific constant that represents the special character. The fault is at line 70, where the branch predicate should be s[*i+1] == ENDSTR, but an incorrect array index, *i, is used instead. Given this faulty program and a set of test cases for the program that trigger the fault (*i.e.,* at least one test in the set fails due to this fault), MintHint would produce the following hint:

Replace s[*i] == ENDSTR by s[*i+1] == ENDSTR

Developers would use this hint as guidance while modifying the original code to arrive at a repair. In this specific example, developers would simply make the suggested change and obtain the repaired

*Now at Walmart Labs

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

version. In the more general case, as we will show in Section 4, hints are not necessarily complete repairs, but rather practically useful suggestions on how to generate such repairs.

To generate hints, MintHint operates in four steps. The *first step* identifies potentially faulty statements by leveraging an existing fault localization technique that requires only a test suite (*e.g.,* [1, 20, 25]). The subsequent steps are performed for each of the identified statements. The *second step* derives a state transformer, that is, a function that (1) is defined for all program states that reach the faulty statement in the given test suite and (2) produces the right output state for each of them. This step leverages dynamic symbolic execution techniques (*e.g.,* [4]). The *third step* explores a repair space and tries to identify and rank, through a statistical correlation analysis [8, 27], expressions in the space that are likely to occur in the repaired statement, using the state transformer derived in the second step. (To the best of our knowledge, this paper is the first one to apply this form of statistical reasoning to programs.) The *fourth step* of the approach synthesizes repair hints by pattern matching [41] the expressions computed in the previous step with those in the faulty statement. Finally, after computing hints for all potentially faulty statements, MintHint ranks the generated hints to help developers prioritize their efforts.

MintHint synthesizes five types of hints that suggest (1) insertion, (2) replacement, (3) removal, or (4) retention of expressions, and (5) combinations of these. As shown in Table 1, these hints are applicable to many types of common faults, including incorrect, spurious, or missing expressions, as well as combinations of these. Moreover, MintHint can handle faults in a variety of program constructs, such as assignments, conditionals, switch statements, loop headers, return statements, and statements with ternary expressions. MintHint's main restriction is that the fault has to involve a single statement and, in the case of an assignment, it must be in the right-hand side expression. However, we believe that MintHint can be extended to address other situations, such as those where the left-hand side variable in an assignment is faulty or the fault spans multiple statements.

MintHint overcomes the limitations of existing techniques, which we listed earlier, in the following ways. First, it does not rely on a formal specification; it instead derives an operational specification (*i.e.,* a state transformer) from the test cases available. Second, approaches that aim at deriving complete repair typically use equality with the state transformer (or an analogous entity) as a criterion for selecting a candidate repair (*e.g.,* [29, 31]). The statistical correlation used in MintHint is a *more relaxed and robust notion than equality* and can thus be more effective in identifying which expressions are *likely* to be part of the repaired code; this allows MintHint to synthesize more general repairs and to be effective in the presence of incomplete data or even imperfect data (*i.e.,* state transformer mappings that do not arise in execution of the fault-free program). Third, since MintHint looks for building blocks of repair (rather than the complete repair itself) and then combines them algorithmically to generate compound hints, it can generate useful, *actionable* hints even when exploring an incomplete repair space.

To evaluate MintHint, we developed a prototype tool that implements our approach for C programs and performed a *user study* using programs from the Siemens benchmark [19]. The study involved 10 users and consisted of a control phase and an experimental phase. In both phases, we provided each user with a single repair task along with fault localization information and a test suite. In the experimental phase, in addition, we gave the users repair hints generated by MintHint. (The tasks we gave to a user in the two phases were independent.) Without repair hints, only 6 of the 10 users completed their task within 2h. With repair hints, *all* 10 users could complete their task within the same time limit. Moreover, the tasks completed in both phases were completed over 5 times faster by the users who used MintHint's repair hints.

Table 1: Nature of hints and targeted faults.

Nature of hint	*Targeted fault*
Insert	Missing expressions
Replace	Incorrect operator, constant, variable, etc.
Remove	Spurious expressions
Retain	Filtering out non-faulty statements
Compound	One or more occurrences of above

In addition, we evaluated MintHint on a total of 11 faulty versions of Unix utilities sed, flex, and grep [12]. Symbolic execution timed out for one of them, and hence we did not generate hints for that version. For 7 of the remaining 10 faulty versions, MintHint generated hints that immediately led to a repair, that is, we did not have to manually identify any additional transformations to achieve repair. In one more case, the hints resulted in a partial repair, in which all passing tests continued to pass, and several previously failing tests passed.

It is worth noting that, both in the user study and in the evaluation on Unix utilities, MintHint was able to synthesize useful hints even in the many cases in which (1) the repair space considered did *not* contain the repaired version of the faulty expression or (2) the state transformer contained imperfect data. In general, a fault localization tool may incorrectly flag a statement as suspicious. MintHint can discover such cases and generate "retain" hints for them, helping developers focus their attention on the other statements. In our experiments, MintHint filtered 40% of such cases altogether.

In summary, the main contributions of this paper are:

- The definition of MintHint, a *novel sophisticated program-repair technique that suitably combines symbolic, statistical, and syntactic reasoning* and overcomes some of the main limitations of existing techniques by focusing on generating repair hints, rather than complete repairs.
- An implementation of MintHint that can perform automated synthesis of repair hints for C programs.
- A user study that evaluates the usefulness of repair hints during debugging. This is *one of the few user studies performed in the area of program repair and debugging in general*.
- A further evaluation of MintHint's effectiveness on several faulty versions of three real-world programs.

2 OVERVIEW

Figure 2 provides a high-level view of MintHint. As the figure shows, MintHint takes as input a faulty program and a test suite—where at least one test case triggers the fault in the program, and thus fails—and produces a list of repair hints in four steps. We now describe these four steps using the example we discussed in the Introduction, shown in Figure 1.

Step 1: Fault localization. This is a preliminary step, whose goal is to provide the hint generation algorithm with a list of possibly faulty statements to be repaired. To compute this list, MintHint can leverage any existing fault localization approach. In our current implementation, we use the Ochiai approach as implemented in the Zoltar tool [20], which performs spectra-based fault localization. For our example, the localization tool computes the following ranked list of suspicious statements: 80, 77, 70, 73, 502, and so on.

In its subsequent steps, MintHint runs its hint generation algorithm on each of the first few statements in the fault localization list *independently*. In the following discussion, we use line 70 to illustrate the remaining steps of MintHint.

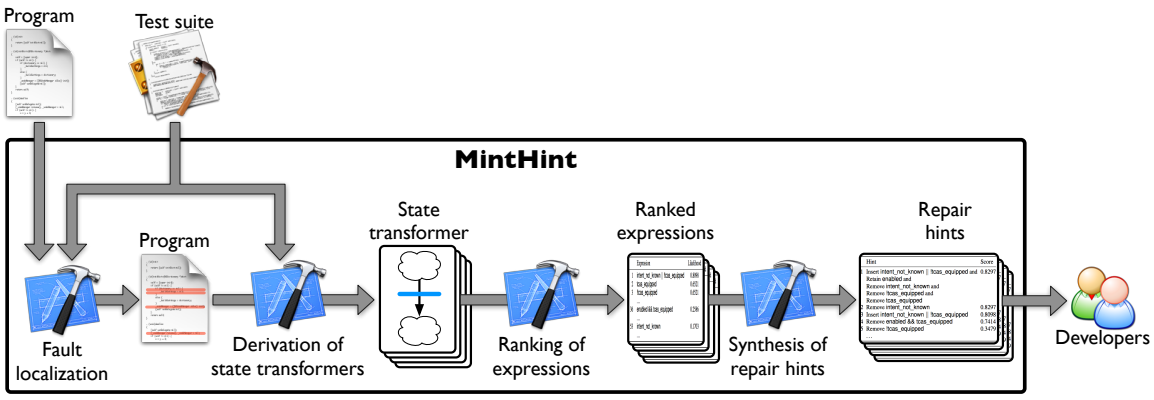

Figure 2: High-level view of the MintHint approach.

Table 2: An example state transformer and values of some expressions over the input states.

Input state		*Output state*	*Values of expressions over input state*		
s	*i	$branch_0$	s[*i] == s[*i+1]	s[*i] == ENDSTR	s[*i+1] == ENDSTR
"@%&a"	0	*false*	false	false	*false*
"@n@"	0	*false*	false	false	*false*
"@n@"	2	*true*	false	false	*true*
"%@@"	1	*false*	true	false	*false*
"V@"	1	*true*	false	false	*true*

Step 2: Derivation of state transformers. This step infers a specification of what would be the correct computation at line 70. To do so, in the absence of an actual specification, MintHint uses the test suite provided for the program to infer an operational specification for the statement in the form of a state transformer. Informally, this *state transformer* is a function that, given an input state (*i.e.,* valuation to variables) at the (potentially) faulty statement, produces an output state that would make every test in the test suite pass, including the failing ones. For passing tests, state transformers can be easily computed by simply observing input and output states during the execution of the tests. For failing tests, as we discuss in detail in Section 3.1, MintHint computes state transformers using dynamic symbolic execution and constraint solving [4].

Let us illustrate this step using the statement at line 70. Because the statement is a conditional statement, MintHint uses a fresh boolean variable, say $branch_0$, to represent the outcome of the branch predicate. That is, the statement is transformed into an assignment $branch_0$ = s[*i] == ENDSTR, in which the fault is on the right-hand side (RHS) expression, and the branch predicate becomes the variable $branch_0$. Table 2 shows some entries in the state transformer of the newly created assignment statement: (1) the string array s and index expression *i in the input state and (2) the left-hand side (LHS) variable $branch_0$ in the output state. The values of all variables except $branch_0$ remain unchanged between a pair of input/output states, and hence are not shown again. The value of $branch_0$ in the first row is obtained from a passing test by concrete execution, whereas the remaining values are obtained from failing tests using symbolic execution. These are the values that $branch_0$ is *expected* to assume in the repaired version, for the respective input states.

Step 3: Ranking of expressions. The goal of this step is to identify *syntactic building blocks* (*i.e.,* expressions) for constructing an RHS expression compliant with the computed state transformer. To do so, MintHint searches the solution space, called the *repair space*, for expressions whose values over the input state of the state transformer are statistically correlated with the corresponding output values produced by the state transformer. More precisely, MintHint interprets *correlation coefficients*, which represent the numerical measure of the strength of a statistical correlation [8, 27], as the *likelihood* of the expression to occur in the repaired RHS. For our example, MintHint would compute correlations between the values of the candidate expressions and the expected value of $branch_0$ at line 70. Table 2 gives the values over the input states for two expressions from the repair space and for the faulty RHS itself. ENDSTR is the null character indicating end-of-string. As the table shows, the values of s[*i+1] == ENDSTR are highly correlated with the expected values of $branch_0$, but the same is not true for the other expressions. Table 3 shows some expressions in the repair space ranked by decreasing value of likelihood.

Table 3: A partial list of expressions ranked by the likelihood of occurrence in the repaired RHS.

Rank	*Expression*	*Likelihood*
1	s[*i+1] == ENDSTR	0.62
2	s[*i+1] <= ENDSTR	0.62
	...	
7	s[*i+1] != ENDSTR	0.62
	...	
23	s[1] == ENDSTR	0.43
	...	
488	s[*i] == ENDSTR	0
	...	

Step 4: Synthesis of repair hints. After producing a ranked list of expressions, MintHint analyzes this list to synthesize an *actionable list* of repair hints. MintHint generates two types of hints: simple and compound. A *simple hint* is a single program transformation, whereas a *compound hint* is a set of program transformations. For our example, the hints synthesized by MintHint for some of the statements identified by the fault localization tool are given in Table 4. Note that MintHint may generate more than one hint for the same statement (*e.g.,* for statement 70 in Table 4). We now explain how MintHint generates such hints.

For **simple hints**, MintHint iteratively selects expressions from the repair space of each statement individually, such that (1) their likelihood values are above a given threshold and (2) the statistical correlation among themselves is low. That is, they form a set of expressions such that *any one* expression in the set is likely to appear in the repaired RHS. In our example, MintHint would select s[*i+1] != ENDSTR and s[1] == ENDSTR. We discuss the algorithm for selection of expressions in detail in Section 3.2.

After selecting these expressions, MintHint *pattern matches* [41] each of the selected expressions with the faulty RHS. Based on the

Table 4: Repair hints synthesized by MintHint.

Rank	*Statement*	*Hint*	*Score*
1	70	**Replace** s[*i] == ENDSTR with s[*i+1] == ENDSTR	1
2	70	**Replace** s[*i] == ENDSTR with s[1] == ENDSTR	1
3	70	**Insert** s[*i+1] <= ENDSTR and **Remove** s[*i] == ENDSTR	1
4	77	**Retain** the statement	1
5	80	**Retain** the statement	1
6	502	**Retain** the statement	1
7	70	**Insert** s[*i+1] != ENDSTR	0.62

edit distance between an expression and the subexpressions in the faulty RHS, MintHint determines the nature of the hint to be generated for the expression. If the edit distance is less than or equal to a threshold chosen heuristically (2 in our current formulation), MintHint suggests the replacement of the matching expression in the faulty RHS. Otherwise, if the edit distance is greater than the threshold, it suggests an insertion. Expression s[1] == ENDSTR, for example, is at edit distance 2 from the faulty RHS's expression s[*i] == ENDSTR, so MintHint synthesizes a replace hint for the RHS (position 2 in Table 4). The edit distance for s[*i+1] != ENDSTR, conversely, is greater than 2, and MintHint synthesizes an insert hint for this expression (position 7 in Table 4).

If the edit distance of an expression selected through likelihood value is zero, that is, it already occurs as a subexpression in the faulty RHS, MintHint deems the RHS's subexpression as *unlikely to be faulty* and generates a retain hint for it. In our example, this happens for lines 80, 77, and 502 (positions 5, 4, and 6, respectively). In fact, there is no way to repair the fault through simple modifications of these statements. Thus, retain hints can help developers localize a fault by eliminating some spurious statements (*i.e.,* false positives) returned by the fault localization tool. Finally, MintHint generates remove hints for subexpressions of the faulty RHS that do not have retain hints or hints suggesting their replacement. In our example, as discussed above, a replace hint was synthesized for s[*i] == ENDSTR and hence no remove hint is generated for it.

MintHint ranks hints based on their *score*. Intuitively, the score indicates the confidence in the applicability of the hint and is derived from the likelihood values of the expressions involved. In the case of a replace hint, the score is the maximum of (1) the likelihood of the expression being used for replacement and (2) one minus the likelihood of the expression being replaced. For the replace hint at position 2 in Table 4, for instance, the score is $\max(0.43, 1-0)$, where 0 is the likelihood value of the faulty RHS that is being replaced (see Table 3). For insert and retain hints, the score is the likelihood of the expression being inserted or retained (*e.g.,* 0.62 for the insert hint at position 7 in Table 4). For a remove hint, the score is one minus the likelihood of the expression being removed.

Whereas simple hints can address faults that can be repaired through a single syntactic transformation, **compound hints** can help repair more complex faults—faults that require either more than one program transformation to be repaired or more refined pattern matching. Further, if the repair space contains only building blocks of the repaired RHS, but not the repaired RHS itself, compound hints—obtained by algorithmically combining these building blocks—can bring the repair hints closer to the actual repair.

MintHint synthesizes compound hints by first computing sets of expressions such that (1) within each set, the likelihood values of the expressions are above a threshold and (2) all expressions in the repair space that are likely to appear in the repaired RHS *together* are included in the same set. This computation uses a variant of correlation coefficients, called *partial correlation coefficients*, and a more refined pattern matching. (We defer the detailed discussion of this part of the technique until Sections 3.1 and 3.2.) In our example, MintHint would compute two such sets: {s[*i+1] == ENDSTR} and {s[*i+1] <= ENDSTR}. The selection criterion used here successfully identifies the required expression s[*i+1] == ENDSTR. Although in this case these sets are singletons, the sets would generally contain more than one expression (*e.g.,* see task 10 in Section 4.1).

After computing a set of expressions that may occur together in a repair, MintHint synthesizes a compound hint using the edit distance between each expression in the set and the faulty RHS. For the first set in our example, {s[*i+1] == ENDSTR}, MintHint generates the replace hint at position 1 in Table 4. For the second set, {s[*i+1] <= ENDSTR}, it generates the hint at position 3 in Table 4, which has two *constituent hints*: an insert hint for the expression in the set and a remove hint. MintHint adds the remove hint because, in this compound hint, there is no constituent hint which suggests retention or replacement of the faulty RHS. The score of a compound hint is the maximum of the scores of the constituent hints.

At this point, developers can manually **apply the hints** produced by MintHint by modifying the potential faulty statement according to such hints. For example, in the case of a replace hint, developers should replace a subexpression with a suggested one. For a remove hint, conversely, developers should remove the subexpression in the hint. Developers would also have to suitably remove the operator(s) around the subexpression, or guess another expression to fill the hole, so as to obtain a well-formed resulting expression. Finally, for an insert hint, developers should combine the subexpression in the hint with the existing expression by selecting an appropriate operator and a place for insertion.

Algorithm 1: Algorithm MintHint

Input: Program P, test suite T, the number of faulty statements k, the bound on the size of expressions in the repair space m
Output: Ranked list of repair hints for the faulty statements

```
1  begin
2    SF ← localize_faults(P, T, k) // Localize faults
3    Hints ← ∅ // Initialize the set of hints
4    foreach F ∈ SF do
       // Derive the state transformer
5      Let F be of the form x := e
6      f ← st_trans(P, F, T)
7      Let f be represented as an array [(σ1, σ1'), ..., (σn, σn')]
       // Enumerate expressions in repair space
8      S ← subexps(e)
9      V ← vars_in_scope(P, F); E ← enum_exps(G, V, m)
       // Generate the data
10     D(x) ← [σ1'(x), ..., σn'(x)]
11     foreach e' ∈ S ∪ E do D(e') ← [σ1(e'), ..., σn(e')]
       // Synthesize hints
12     Hints ← Hints ∪ MintSimpleHints(F, D, S, E)
13     Hints ← Hints ∪ MintCompoundHints(F, D, S, E)
14   end
15   return sort(Hints)
16 end
```

3 ALGORITHM

Algorithm 1 shows the MintHint algorithm. As we discussed in the previous section, the inputs to the algorithm are a program P and a test suite T such that the program fails on at least one of the tests in T. MintHint also takes a threshold k on the number of faulty statements to be considered for repair and a bound m on the size of the expressions in the repair space.

The first step of MintHint (line 2) is to use the test suite to perform fault localization, as indicated by the function localize_faults,

which produces a list SF of up to k statements ranked by their likelihood of being faulty. All statements in SF are then processed in a loop (lines 4–14). Function st_trans at line 6 computes the state transformer for a faulty statement F. Sets S and E, computed at lines 8 and 9, contain the subexpressions of the faulty RHS and the extra expressions that should be considered when searching for repairs. The function D maps expressions from the repair space and the LHS variable to their values according to the state transformer (similar to the column values in Table 2). Two separate algorithms, MintSimpleHints and MintCompoundHints, are used for synthesizing simple and compound hints. Finally, the hints across all statements are sorted by their scores (line 15).

In the following discussion, for simplicity, we consider statement F to be of the form $x := e$. In step 2 in Section 2, we discussed a transformation of the conditional to the form above. MintHint can also handle cases in which the potentially faulty statement is a loop header of the form for(init, cond, upd), where init initializes the loop counter(s), cond is the loop termination condition, and upd is the update of the loop counter(s). In these cases, the fault could be in any of these three components, so MintHint spawns three different tasks for init, cond, and upd. MintHint treats an assignment with a ternary RHS expression as a conditional statement in which the fault is in the branch predicate or in one of the assignments in the branches. MintHint can handle other constructs, such as switch or return statements, in analogous ways.

3.1 From Tests to Likelihood of Expressions

The first part of the MintHint algorithm is based on *symbolic and statistical analysis* and is performed in several phases. We discuss each of these phases separately.

3.1.1 Fault Localization

Function localize_faults leverages an existing fault localization technique to compute a set SF of potentially faulty statements. Any fault localization technique that requires only the program P and a test suite T can be used here (*e.g.,* [1, 20, 25]). Analogously to existing repair approaches (*e.g.,* [29, 31]), MintHint assumes that faults can be repaired by changing a single statement. Therefore, MintHint synthesizes repair hints for each statement F in SF independently.

3.1.2 Derivation of State Transformers

For each potentially faulty statement F, function st_trans derives a *state transformer* f that, when substituted for F, makes the program produce the correct output for each test in T. f is a function from program states to program states, where a *program state* σ is a mapping from the variables in scope at the faulty statement to appropriately typed values. More formally, in Algorithm 1 f is an array of pairs of input/output states: $f = [(\sigma_1, \sigma_1'), \ldots, (\sigma_n, \sigma_n')]$ (line 7). Notationally, an unprimed state is an input state (at F), and a primed state is the corresponding output state.

Function f is defined for states that can be witnessed at F, given the inputs in T. Tests in T that do not execute F are ignored when computing state transformers. For each passing test traversing F, MintHint runs the program and collects the input/output states at F. Conversely, for failing tests that traverse F, MintHint (1) makes the LHS variable x symbolic, (2) uses a *symbolic execution algorithm with constraint solving* to obtain values of x that makes the program produce the correct output, and (3) reruns the program concretely using the so computed values for x (instead of the original values of the RHS). The input/output states at F in this concrete execution give the mapping f for the failing test.

MintHint sets a timeout for symbolic execution of each failing test. Symbolic execution may fail to obtain values for x that result in the correct output either because it times out or because of constraint unsatisfiability. If the symbolic execution times out on *every* failing test of F, MintHint discards the statement and does not generate any hints for it. If the constraint obtained by making x symbolic is unsatisfiable for *every* failing test of F, MintHint generates a "retain the statement" hint. Intuitively, since the program fails irrespective of F (as evidenced by unsatisfiability of the contraints), F may *not* be the faulty statement.

In its subsequent phases, MintHint treats the state transformer f for a potentially faulty statement F as an *operational specification* for repair, thus eliminating the need to have a formal specification.

3.1.3 Ranking of Expressions

Using the state transformer f, MintHint ranks expressions in the repair space of F according to their likelihood of occurring on the RHS of the repaired version of F. The repair space can be obtained in several ways, such as by enumerating expressions over variables in scope or mining expressions that occur elsewhere in the program (similar to what is done in [29]). Presently, MintHint uses the former approach. More precisely, function vars_in_scope, in Algorithm 1, computes the set V of variables in scope at F. Then, function enum_exps enumerates the expressions of size up to m (a user-defined threshold) over V. Let E be the set of these expressions. MintHint also includes the set of subexpressions of the RHS expression e (including e itself) in the repair space. It does so because it evaluates whether they are likely to be faulty or not *independent* of the fault localization results. This set, which we call S, is computed by function subexps (line 8). The repair space is thus $E \cup S$.

We recall that a program state σ maps variables to values. This mapping can be extended naturally to expressions. If $e \equiv e'$ *op* e'' then $\sigma(e) = [\![op]\!](\sigma(e'), \sigma(e''))$. Let $D(x) = [\sigma_1'(x), \ldots, \sigma_n'(x)]$ be the values of the LHS variable x over the output states defined by state transformer f (line 10). Similarly, for an expression e' in the repair space, let $D(e') = [\sigma_1(e'), \ldots, \sigma_n(e')]$ be the values of the expression e' over the input states defined by f (line 11).

Given the data $D(x)$ and $D(e')$, MintHint's goal is to find whether x and e' are related with each other, that is, whether a change of value in one is accompanied by a change of value in the other. Because $D(x)$ contains the *expected (correct)*, rather than current (erroneous), values of x for the failing tests, the faulty RHS and its faulty subexpressions should *not* be highly related with x. Therefore, the expressions that are highly related with x will be treated as building blocks for the repair. Of course, just because the fault localization tool marks a statement as potentially faulty, it does not mean that the statement is actually faulty. The statistical analysis performed by MintHint is *discriminative* enough so that, for a spuriously marked statement (*i.e.,* a false positive), the existing RHS expression e itself may appear as highly related to x.

Statistical Correlation. In statistics, the problem of whether two statistical variables are related with each other is solved by *statistical correlation analysis*, and a *correlation coefficient* gives a numerical measure of the strength of the correlation between two variables [8, 27]. In our context, a statistical variable is either the LHS variable x or an expression e'. The (absolute) value of a correlation coefficient ranges over $[0, 1]$. A correlation coefficient value close to 0 indicates that the variables are statistically uncorrelated, whereas values close to 1 indicate strong correlation.

There are a number of coefficients that are used for identifying different types of correlations between variables (*e.g.,* linear or monotonic correlations). In its current form, MintHint uses two coefficients: the Spearman coefficient and the Spearman partial correlation coefficient [8]. The former can be applied to any data domain for which there is a ranking function r that can map the values in the data domain to a totally-ordered set. The latter is a variation of the Spearman coefficient to compute the strength of the correlation between a pair of variables by eliminating the effect of a *controlling set* (*i.e.,*

Algorithm 2: Algorithm MintSimpleHints

Input: The faulty statement $F \equiv x := e$, a mapping D from expressions to data, the set S of subexpressions of the faulty RHS, the set E of enumerated expressions
Output: A set of simple hints

```
1  begin
2    R ← S ∪ E; SH ← ∅; L ← ∅
3    while true do
       // Select the most likely expression
4      e′ ← argmax_{e″∈R} likelihood(e″); R ← R \ {e′}
       // Exit loop if likelihood below threshold
5      if likelihood(e′) ≤ δ then break
       // Ensure that e′ is not subsumed by L
6      if p_likelihood(e′, L) ≥ β then
7        L ← L ∪ {e′}
         // Synthesize a simple hint using e′
8        (e″, dist) ← MinEdit(e′, S)
9        SH ← SH ∪ GenHint(e′, e″, dist)
10     end
11   end
     // Remove-hints for unlikely expressions
12   foreach e′ ∈ S for which there is no retain/replace hint do
13     SH ← SH ∪ {(Line(F), Remove e′, 1 − likelihood(e′))}
14   end
15   return SH
16 end
```

a subset of the other variables). The complexity of computing the Spearman coefficient between datasets of size n is $O(n \log n)$. The complexity of computing the Spearman partial coefficient is $O(n^3)$, for n larger than m, where m is the size of the controlling set.

Likelihood and Ranking of Expressions. The MintHint algorithm is based on the **hypothesis** that an expression e' is likely to occur on the RHS in the repaired version of F iff it is highly correlated with x on the dataset obtained from the state transformer which gives the expected, correct values of x even for the failing tests.

The *likelihood* of an expression e' to occur in the repaired RHS, denoted by likelihood(e'), is the absolute value of the Spearman coefficient between e' and x over the datasets $D(e')$ and $D(x)$ (obtained at lines 10 and 11). Given a set L of expressions, the *partial likelihood* of e' to occur in the repaired RHS *along with* the expressions in L, denoted by p_likelihood(e', L), is the absolute value of the Spearman *partial* correlation coefficient of $D(e')$ and $D(x)$ with $\{D(e'') \mid e'' \in L\}$ as the data of the controlling set L.

3.2 From Likelihood of Expressions to Hints

In this second part, we discuss how MintHint utilizes the likelihood values of expressions computed in the first part of the algorithm to synthesize hints. Intuitively, MintHint synthesizes hints by performing *syntactic pattern matching* between the expressions that are likely to occur in the repaired RHS and the subexpressions of the faulty RHS, using different patterns to address different types of possible faults. Formally, a *repair hint* h for a statement F is a triple (ℓ, t, s) comprising the line number ℓ of statement F, a textual hint t, and the hint's score s. The keywords in the textual hint, shown in bold font, have their usual English meaning.

3.2.1 Simple Hints

MintSimpleHints (see Algorithm 2) synthesizes a set SH of simple hints given a faulty statement F, a dataset D obtained from F's state transformer, subexpressions S of the faulty RHS, and extra expressions E that constitute repair candidates. The algorithm also makes use of two thresholds, δ and β, to select expressions by likelihood and check partial likelihood of a candidate expression given the already selected expressions, respectively.

MintSimpleHints initializes the repair space R to $S \cup E$, and the set of simple hints SH and the set of likely expression L to the empty set (line 2). It then executes the loop starting at line 3 until the likelihood of expressions drops below the threshold δ (line 5). Within this loop, an expression $e' \in R$ with the highest likelihood is selected and removed from R (line 4). If the partial likelihood of e' with L as the controlling set is above the threshold β (line 6), it is added to L. This check ensures that e' has sufficient statistical correlation with x (the LHS) after taking out the effect of L (*i.e.,* e' is not subsumed by L). In the example in Section 2, once s[*i+1] != ENDSTR is added to L, all expressions up to s[1] == ENDSTR (ranked 23^{rd}) have partial correlation below the threshold $\beta = 0.1$.

At this point, MintSimpleHints generates a simple hint for e'. To do so, it first invokes function MinEdit, which identifies the expression e'' from S with minimal edit distance $dist$ from e' [41] (line 8). Then, the algorithm invokes function GenHint, which synthesizes a simple hint based on the value of $dist$ (line 9). If $dist$ is below a given threshold (but $dist > 0$), the algorithm generates hint "**Replace** e'' by e'". Conversely, if $dist$ is above such threshold, it generates hint "**Insert** e'". For the former, the score of the hint is set to the maximum between the likelihood of e' and the *un*likelihood of e'' (*i.e.,* $1 - \text{likelihood}(e'')$). For the latter, the score is simply the likelihood of e'. In its current formulation, MintHint uses a threshold of 2 over the edit distance. In our empirical evaluation, it captures errors resulting from incorrect operator, constant, variable, array index, and so on. Increasing the threshold may generate spurious replace hints, whereas reducing it would produce two separate hints, insert and remove, for e' and e'' respectively.

If $dist = 0$ (*i.e.,* e' already belongs to S), the algorithm generates hint "**Retain** e'" and assigns to the hint a score equal to likelihood(e'). A special case of this is when e' is exactly the same as the RHS expression for which the algorithm generates a "retain the statement" hint. Finally, for the expressions that appear in the faulty RHS but do not have an exact or close match (with $dist$ less than the threshold) in set L, the algorithm generates remove hints in the loop at line 12, where function Line returns the line number of a statement. The score of a "**Remove** e'" hint is $1 - \text{likelihood}(e')$.

3.2.2 Compound Hints

Function MintCompoundHints, called in Algorithm 1 at line 13, computes compound hints using the same inputs as MintSimpleHints. Due to space constraints, we do not typeset MintCompoundHints and simply explain the key similarities and differences between Algorithm 2 and MintCompoundHints. Similar to Algorithm 2, it iteratively computes the set of likely expressions. However, it uses p_likelihood(e'', L) in place of likelihood(e'') at line 4, and the branch predicate at line 6 is replaced with $true$. The value of partial likelihood gives the measure of the likelihood of e'' to appear in the repaired RHS *along with* the expressions in L. Consequently, the expressions added to set L are all those expressions that can occur *together* in the repaired RHS. In contrast, the selection of an expression at line 4 in Algorithm 2 is based on its *individual* likelihood.

In general, MintCompoundHints can generate more than one set of likely expressions. More specifically, if multiple expressions have the highest partial correlation coefficient (line 4) in the first iteration of the loop, the algorithm would partition them into three sets based on their minimum edit distance from the faulty RHS and its subexpressions: (1) equal to zero, (2) less than equal to 2, and (3) more than 2. The algorithm initializes three sets of likely expressions by selecting one expression from each of the partitions above (if not empty). It then proceeds independently to select other expressions to add to each of them. (Conversely, when generating simple hints for multiple expressions with the highest correlation coefficient, MintSimpleHints selects one expression at random without partitioning the expressions by edit distance.) For example, in Section 2, the expression s[*i+1] == ENDSTR is the only expression,

Table 5: Description of tasks.

Program	*LOC*	*#Tasks*
print_tokens2	570	2
replace	564	4
tcas	173	4

among the seven expressions with the highest likelihood, that is at edit distance less than equal to 2.

For each expression in set L, MintCompoundHints generates a hint with the pattern matching logic used at line 9 in Algorithm 2. We call each of these hints a *constituent hint* of the compound hint. We say that a pair of constituent hints of a compound hint *conflict* if they refer to either the same or overlapping subexpressions of the faulty RHS. Two subexpressions of the faulty RHS overlap if their subtrees in the AST of the faulty RHS have some common node(s), which can be checked by performing a simple walk over the AST. In order to ensure that no conflicting hints are generated, MintHint removes a subexpression and all the expressions that overlap with it from the repair space as soon as the subexpression is added to set L. (Simple hints are independent of each other and hence are permitted to conflict.) Finally, analogous to the loop at line 12 in Algorithm 2, MintCompoundHints generates remove hints for the subexpressions of the faulty RHS which do not have an exact or close match in L.

There could be multiple simple hints that suggest retention of different subexpressions of the RHS. In these cases, these hints are clustered into one single compound hint. The score of a compound hint is the maximum of the scores of the constituent hints.

4 EVALUATION

To assess the effectiveness of our approach, we implemented the MintHint algorithm for C programs and performed an empirical evaluation. Our implementation leverages Zoltar [20] for fault localization, KLEE [4] for dynamic symbolic execution with constraint solving, and Matlab[1] for statistical analysis.

Our evaluation consists of two main parts: (1) a *user study* that assesses whether MintHint can improve developers' productivity, and (2) an empirical study in which we apply MintHint to several faulty versions of three *Unix utilities* to further assess the effectiveness of the approach. Specifically, we investigated the following questions:

RQ1: *Usefulness of hints* — Can MintHint produce useful hints, thus enabling developers to repair programs more effectively?

RQ2: *Robustness* — How does MintHint perform when the repair space is incomplete (*i.e.,* does not contain the repaired version of the faulty expression) and is supplied imprecise data?

RQ3: *Performance and scalability* — How well does MintHint scale to large programs, state transformers, and repair spaces?

4.1 User Study

Experimental Setup. We performed fault localization on a set of programs from the Siemens suite [19], which consists of programs with multiple faulty versions. Table 5 lists the programs and the number of faulty versions that were selected as tasks. The tasks represent a diverse collection of faults (see Table 6). In the user study, each user was required to work on two independent tasks. To keep each task manageable within 2h, we presented to the user only the top 5 statements identified by Zoltar as potentially faulty. For each of the chosen tasks, the actual faulty statement belonged to this list. For each program and candidate faulty statement, MintHint obtained the state transformers for the failing tests through symbolic execution with a timeout of 5m per test. For one of the candidate tasks, replace-v18, symbolic execution of many failing tests timed out. In comparison, there were many more passing tests—potentially making data from failing tests statistically insignificant. To avoid this, only half of the passing tests were used for deriving the state transformer.

Of the 10 users who participated in our study, 8 were working professionals and 2 were graduate students (with prior industry experience) — none affiliated with our research group. Of these, 8 stated that they had moderate to high expertise with C programming/debugging, and all 10 had at least 1 year of experience with C programming. For each task, the input/output specifications of methods, together with the meaning of variables and named constants, were presented as comments in the source code. In addition, each user was provided with a test suite of 10 passing tests and was given 15m to study the program before starting to repair it.

We performed the user study in two phases. In the *control phase*, the users were given the fault localization information and the test suite. In the *experimental phase* they were also given the repair hints. Each user worked on a single task per phase and was given 2h to complete that task. We considered a task to be *complete* if the repaired program passed all the tests. The users chose the programs for the control phase by drawing lots. We mapped each task in the control phase to a task in the experimental phase to make sure that a user would not work on the same task, or on another faulty version of the same program, in both phases.

(RQ1) Usefulness of hints. Table 7 summarizes the results of the user study. In the control phase (*i.e.,* without hints), the users could localize the fault in $8/10$ cases, but only managed to repair the programs in 6 cases. On the contrary, in the experimental phase (*i.e.,* with hints), the users were able to perform localization and repair in all 10 cases. Further, the average time taken to repair a fault was 91m in the control phase (except for the 4 timeouts), whereas it was 29m in the experimental phase. The average speedup obtained with the use of hints, for the 6 tasks that were completed in both phases, was 5.8x. We asked the users to rate the difficulty level of localization and repair for their tasks as easy, moderate, or difficult. The ratings, *for the same set of tasks*, differed across the two phases. Notably, with hints, 4 more tasks were rated by the users as easy. These *qualitative ratings* corroborate the quantitative results presented above.

The users uniformly reported that the hints were useful and were asked to indicate the most useful hint. In Table 6 we report, for each task, the rank of the most useful hint (as identified by the user) over the entire list of hints presented to the user for that task. As the table shows, the most useful hints were all in top 10 except for one case. The last column in Table 6 shows the number of statements for which *only* a "retain the statement" hint was generated. These are the statements that are classified by MintHint as unlikely to be faulty. Across the 10 tasks and 50 statements in the fault localization lists, 40 statements are likely to be false positives, as only 10 are true positives (*i.e.,* definitely faulty). Out of these 40, MintHintfilters out 11 statements (over 25%). This contributes greatly to ease of localization. As further evidence that there is no obvious way to rectify these 11 statements, we note that no user in the study was able to produce a repair for any of them.

For 6 tasks (tasks 2–6 and 9), the most useful hint had the *precise* information required to repair the fault. For task 10, the faulty RHS was of the form exp1 || exp2, and the compound hint suggested that both exp1 and exp2 be retained, but it did not report the same for the entire RHS. The user therefore suspected that the operator was incorrect and correctly replaced it with &&. For Task 1, the hint suggested only removal of the incorrect expression, so the user had to produce a substitute expression. In Task 7, the hint did not suggest the expression to be inserted. In fact, the user mentioned that the hints helped mainly in localizing the fault. This is possible because MintHint eliminates 2 other statements in this case (see the last column for Task 7 in Table 6). In the case of Task 8, the replace hint did not

[1] http://www.mathworks.com/products/matlab/

Table 6: Description of faults in the user study and of useful hints identified by the users: The tasks that could not be finished in the control phase (without hints) are underlined. All the tasks were completed in the experimental phase (with hints).

Task	*Program-Version*	*Nature of fault*	*Type of the most useful hint*	*Rank of the most useful hint*	*#Total hints*	*#Stmts with only "Retain the stmt" hints*
1	print_tokens2-v6	Incorrect array index	Remove	10	31	–
2	replace-v7	Superfluous expression	Compound (Remove + others)	4	13	1
3	print_tokens2-v7	Superfluous expression	Compound (Remove + others)	4	8	–
4	replace-v18	Missing expression	Compound (Insert + others)	1	3	1
5	replace-v23	Incorrect array index	Replace	1	6	3
6	tcas-v28	Incorrect operator	Replace	6	26	–
7	replace-v8	Missing expression	Compound (Retain + others)	1	6	2
8	tcas-v2	Incorrect constant	Replace	7	13	1
9	tcas-v1	Incorrect operator	Replace	2	11	3
10	tcas-v12	Incorrect operator	Compound (Retain + others)	11	14	–

Table 7: Results of the user study.

	Control phase (without hints)	*Experimental phase* (with hints)
Quantitative Analysis		
Successful localization	8/10	10/10 (+2)
Successful repair	6/10	10/10 (+4)
Avg. time to repair	91m + 4 timeouts	29m (no timeouts)
Avg. speedup (excl. timeouts)	NA	5.8x
Qualitative Analysis (Ratings given by the users)		
Difficulty of localization	Easy: 2 Moderate: 6 Difficult: 2	Easy: 6 (+4) Moderate: 3 Difficult: 1
Difficulty of repair	Easy: 3 Moderate: 5 Difficult: 2	Easy: 7 (+4) Moderate: 1 Difficult: 2

Table 8: Count of tasks in the user study wrt incompleteness of the repair space and noise in state transformers.

	Yes	*No*
Incompleteness of repair space	5	5
Noise in state transformers obtained from tests	7 (max. 27%)	3

suggest the required (named) constant to be used. Nevertheless, the user observed that after substituting the new expression suggested in the hint, many failing tests started passing, and subsequently inferred the right constant manually. Of these, tasks 5, 8, 9, and 10 were *not* completed in the control phase.

The number of total hints per task given in Table 6 depends on several factors. First, the thresholds on correlation coefficients determine how many expressions end up in the set of likely expressions, and thus also affect the number of hints. Across all tasks, the thresholds on correlation coefficient and partial correlation coefficient for generation of simple hints were 0.4 and 0.1, respectively. For compound hints, the threshold was 0.6. Second, if the symbolic execution times out on all failing tests for a statement, the hint generation algorithm is not run for that statement (see Section 3.1.2).

Though the study involves a relatively small number of users, it provides substantial evidence that repair hints are useful in obtaining repairs and also in reducing the time taken for repair.

(RQ2) Robustness. In practice, it is difficult to estimate the syntactic space to search for a complete repair. In our experiments, for each statement, apart from the subexpressions of the potentially faulty expression, expressions of size up to 4 (over the variables in scope at the statement) were added to the repair space. The size of an expression is the number of nodes in its abstract syntax tree (AST). For expressions involving arrays, an occurrence of an array expression is counted as size 1 and the index expressions themselves can go up to size 4. We call a repair space *complete* when the repaired version of the expression belongs to it. For example, in task 10, the repaired expression exp1 && exp2 was not in the repair space.

Table 9: Performance and scalability chart: Tasks from the user study belong to all cells except cell at position $(3, 2)$. Tasks related to Unix utilities (Section 4.2) belong to the shaded cells.

Size of state transformer (#Rows of the data matrix) \ *#Exprs in repair space* (#Columns of the data matrix)	*Upto* 5k	*Upto* 10k	> 10k
Upto 1k	< 1m	< 1m	< 1m
Upto 10k	< 1m	< 5m	< 5m
> 10k	< 1m	< 5m	≈ 1h

MintHint derives state transformers for failing tests by symbolic execution. In some cases, the derived constraints may have multiple satisfying assignments but not all of them can be observed in the execution of the repaired version of the program. The constraint solver may pick any one of them. For passing tests, the state transformer is obtained by concrete execution. Even though the test passes, the value generated by the faulty expression may not be observed in the repaired version. These situations make the resulting data, which is used as a specification, imperfect (noisy) and may in general invalidate the applicability of a repair.

Table 8 gives the count of tasks which had noise in the state transformers and where the repair space was incomplete. This information is provided only for the actual faulty statement. To estimate the amount of noise in the data, we first obtain the *noise-free* state transformer of the (known) repaired version independently by concrete execution over the test suite. An entry in the state transformer obtained over the faulty version is classified as *noisy* if it does not belong to the noise-free state transformer. There were 7 tasks with noisy data with the maximum of 27% noise in one of them and there were 5 tasks where the actual repaired expression did not belong to the repair space. Nevertheless, the successful completion of the tasks in the user study indicates that useful hints could be synthesized even in these challenging cases. The key reasons for this are (1) the use of statistical correlation which is robust in presence of noise and (2) the ability of MintHint to synthesize compound hints from building blocks. In particular, the repair spaces were incomplete for tasks 2–4, 7, and 10, and as Table 6 shows, in each of these cases, the most useful hint was a compound hint.

(RQ3) Performance and scalability. The complexity of statistical correlation computation dominates the cost of hint generation (see Section 3.1.3 for discussion on its complexity). It works on a two-dimentional matrix where the number of rows is equal to the size of the state transformer (the number of input/output pairs) and the number of columns is equal to the number of expressions in the repair space. Table 9 gives the performance and scalability chart

Table 10: Description of faults and hints for Unix utilities.

Task	Version-Fault	Nature of fault	Type of useful hint	Rank of useful hint	#Total hints	#Stmts with "Retain stmt" hint only
Tasks – flex						
1	f2-f2	Incorrect constant	Replace	1	1	0
2	f2-f14	Incorrect operator	Replace	1	1	0
3	f4-f15	Incorrect operator	Replace	1	3	0
Tasks – grep						
4	g3-f10	Incorrect operator	Compound	1	37	14
5	g4-f10	Incorrect operator	–	–	14	14
Tasks – sed						
6	s2-f1	Incorrect operator	Replace	1	11	10
7	s3-f4	Incorrect constant	–	–	11	3
8	s3-f6	Incorrect constant	Replace	1	10	9
9	s5-f1	Incorrect operator	Replace	13	13	12
10	s6-f1	Incorrect *operators*	Compound	2	4	0
11	s6-f2	Symbolic execution times out with 15m threshold				

summarizing all runs of the hint generation algorithm for all tasks and faulty statements in the user study. Despite the large datasets, the hint generation algorithm scales well. Zoltar took slightly over 2m on an average for fault localization. Except for a few statements, KLEE ran to completion on all the failing tests within 5m (the timeout set by us). The timings are measured on a desktop with Intel i5 CPU@3.20 GHz and 4GB RAM.

4.2 Hint Generation for Unix Utilities

Experimental Setup. We applied MintHint on three commonly used Unix utility programs - flex, grep, and sed obtained from the SIR repository [12]. These are reasonably large programs ranging from 10K to 14K lines of code. The SIR repository consists of several versions of these programs seeded with different faults. We performed fault localization on the different versions using Zoltar. We selected those versions for which Zoltar identified the actual faulty statement within top 15 statements in its list. Table 10 lists the versions and fault-IDs of the programs that were selected as tasks. The top 15 statements identified by Zoltar as potentially faulty were considered for repair. For each version and candidate faulty statement, the state transformers were obtained for failing tests through symbolic execution with a timeout of 15m, with the exception of tasks 6, 7 and 8 for which a timeout of 5m was sufficient. In Task 11, symbolic execution timed out for all failing tests and hence it was not subjected to further analysis.

For grep, we observed that the input files supplied along with the program were too large (10K lines) due to which the symbolic execution would result in timeout. Consequently, we isolated the failure-inducing part of the input files. For each failing test, we split the input file into segments comprising 500 lines each. We then ran the test concretely with each segment as the input file and compared its result with the result of the known correct version of grep on the same segment (and the same regular expression). We identified the segment on which the two outputs differed. In all the cases, there was only one such segment. The original failing test was replaced with the segment and corresponding desired output.

(RQ1) Usefulness of hints. For each of the tasks, Table 10 shows the nature of the fault and the type of the most useful hint. For all tasks in flex and tasks 6, 8, 9 in sed, a replace hint synthesized by MintHint when applied, immediatedly lead to success on both passing and failing tests. For task 4 in grep, MintHint synthesized a compound hint for the faulty statement, which suggested removing several expressions from the RHS and retaining a single non-faulty expression. We manually removed the suggested expressions and operators around them, retaining a single well-formed expression. With this change, all the tests passed successfully. Task 10 in sed consists of two faults in the same statement. A compound hint suggested removal of two subexpressions and retention of the non-faulty subexpression. Applying this hint however lead to only a *partial repair* since MintHint did not generate the expressions that should be used for replacing the faulty ones. With this change, all the previously passing tests continued to pass and several previously failing tests too started to pass. For task 5, the symbolic execution timed out on all failing tests for the faulty statement. Thus MintHint could not produce any hints for it. For the other statements identified by Zoltar as potentially faulty, MintHint suggests to retain each of them as they are. Even though we did not obtain a repair for the fault in this task, the retain hints help filter out the non-faulty statements. In task 7, MintHint did not produce any useful hint for the faulty statement (due to excessive noise, as discussed below).

Interestingly, MintHint generated only "retain the statement" hints for many statements identified by Zoltar – effectively filtering them out as non-faulty. Across the 10 tasks and 150 statements in the fault localization lists, 140 statments are likely to be false positives. Out of these, 62 statements (more than 44%) are filtered out by MintHint. In fact, in tasks 6, 8 and 9, retain were the only other type of hints apart from the replace hint required for repair. We studied the faulty statements which had only the retain hints but could not identify a way to change them to repair the faults.

This study gives strong evidence that MintHint can be applied to large programs with not so accurate fault localization lists.

(RQ2) Robustness. Repair spaces were constructed in a manner similar to the user study and with the same bound on expression sizes. The repair space searched by MintHint contained the repaired version of the faulty RHS in all but two cases. The data obtained for the faulty statement from the failing tests contained noise in 4 cases. Our approach for estimating the amount of noise is explained in Section 4.1. In one case (task 7), the noise was 97% and MintHint could not produce any useful hint. Note that we could not measure noise for one of the tasks (task 5) as the symbolic execution did not generate any data for the faulty statement.

(RQ3) Performance and Scalability. In Table 9, the shaded cells denote the time taken by MintHint for statistical correlation analysis and hint generation for the tasks from Unix utilities. No task took more than 5m for these steps. In particular, correlation analysis and hint generation for tasks from flex and grep took less than 10s. Fault localization finished within 1m on an average for sed tasks, 2m for grep tasks, and 3m for flex tasks.

Although symbolic execution finished within 5m for tasks 6–8, it required at least 15m for the remaining tasks. Even with 15m threshold, it timed out for all failing tests for the actual faulty statement in task 5 and for all potentially faulty statements in task 11.

4.3 Limitations and Threats to Validity

One of the main limitations of our approach is its reliance on symbolic execution for deriving state transformers, which is a complex and expensive technique. However, these techniques are becoming increasingly efficient, and many of their practical limitations are being addressed (*e.g.,* [13, 36]). Moreover, as we discuss in Section 6, we plan to investigate alternative, less expensive ways to build state transformers. Further, it is technically difficult to obtain only those values which can be observed in the repaired program through symbolic execution. This makes the state transformers noisy. Due to the statistical reasoning applied in MintHint, it produced useful hints even in presence of noise in many cases. The measurement errors leading to noisy data are common in other domains as well and a large body of work, called *outlier detection*, exists to deal with them

(see [5] for a survey). We plan to investigate applications of these techniques to further improve tolerance of MintHint to noise.

Like every empirical evaluation, ours too has potential threats to validity. Threats to *internal validity* for the user study include *selection bias* where the users working on the same task in control and experimental phases may have different expertise and *testing bias* where activities before the study may affect the outcome. Only two users had indicated low expertise with C programming, to prevent selection bias, we paired them in such a way that their tasks were interchanged in the two phases. Since the control phase was conducted before the experimental phase for all users, it is likely that the users became better accustomed to the debugging task. We mitigated this possibility by ensuring that no user worked on the same task or two faults of the same program, in the two phases of the user study. Further, we only made a presentation about the meaning of repair hints to them and did not provide any hands-on tutorial. There may be faults in our implementation that might have affected our results. To address this threat, we manually checked many of our results and did not encounter any error.

Threats to *external validity* arise because our results may not generalize to other group of developers (in the case of user study) and program repair tasks. In the user study, we ensured that the users did not have any prior experience with the programs used as repair tasks. While this ensures a level playing field, it leaves out users who might have better familiarity with the programs. The tasks in the user study were *not* hand-picked. We applied a well-defined criterion for their selection. The programs in Siemens suite were sorted by the rank of the actual faulty statement in the respective localization lists. Within each rank, the program-fault names were then sorted in the lexicographic order and finally, the first two tasks at each fault localization rank were selected. Similarly, the tasks from the Unix utilities consist of all tasks from the SIR repository [12] with only a single faulty statement such that the statement occurs within top 15 statements returned by a third-party fault localization tool, Zoltar. A few tasks could not be included because of limitations of the symbolic execution tool. The performance of MintHint is a function of the test suite also. It will take a much larger evaluation to ascertain how the quality of tests affects the quality of repair hints. It is however important to note that the tasks and test suites we used were also used in numerous previous papers in the area (*e.g.,* [26, 31, 43]). Nevertheless, the number of tasks considered is small, so our findings may not generalize to other programs or faults.

5 RELATED WORK

Early work by Arcuri [2], later extended by Le Goues and colleagues [29], proposes the use of genetic programming to automatically generate repairs that make failing test cases pass and do not break any passing test case. Debroy and Wong propose a similar approach, but based on the use of mutation [10].

Other approaches rely on the use of program specifications to repair the code. Pei and colleagues propose an approach for finding program repairs given program contracts [32, 40]. Jobstmann, Griesmayer, and Bloem use a game theoretical approach for identifying repairs that satisfy a linear-temporal-logic specification [24]. Gopinath, Malik, and Khurshid's approach builds a SAT formula that encodes the constraints imposed by the specification on the program behavior and, if the formula is satisfiable, derives a repair from the SAT solution [14]. Konighofer and Bloem present a template-based approach that, given a faulty program and a specification, performs a symbolic analysis of program inputs and template parameters to generate a repair [28]. He and Gupta propose a technique that computes the weakest preconditions along a failing trace and compares the computed conditions with functions' pre- and post-conditions to find and correct faults [18]. Logozzo and Ball's approach generates verified program repairs from failed verification checks of programs that have developer-supplied modular specifications [30].

Yet other approaches leverage the existing test suite to infer specifications and generate repairs accordingly. PACHIKA [9] models a program's behavior for passing and failing test cases and generates a repair based on the differences between these models of correct and incorrect behavior. In a recent paper, Nguyen and colleagues propose the SemFix approach [31], which combines angelic debugging [6] and program synthesis [22] to automatically identify program repairs. BugFix [21] shares with our approach the idea of deriving "bug-fix suggestions", rather than actual repairs. Unlike our approach, BugFix generates suggestions using a machine-learning approach based on knowledge acquired from previous bug repairs.

Finally, some approaches perform program repair in specific domains, such as repairs for data structures [11], web application repairs [35], and repairs targeted at security vulnerabilities [33] or concurrency faults [23].

Unlike MintHint, most of these program repair approaches suffer from one or more of the limitations that we discussed in the Introduction. They depend on the presence of a specification, tend to find repairs that are overfitted to a specific set of tests, or must perform a search over a solution space that is large enough to include the (unknown) repair—and are therefore unlikely to be effective in the case of non-trivial repairs.

6 CONCLUSIONS AND FUTURE WORK

We presented MintHint, a novel technique for semi-automated program repair. The key novelty of our approach is that it does not generate complete repairs, an elusive goal in many practical cases, but rather synthesizes repair hints—expressions that are likely to occur in the repaired code. To do so, given only the faulty program and a test suite, MintHint suitably combines symbolic, statistical, and syntactic reasoning. Our evaluation of MintHint provides initial but strong evidence that our approach is effective and practically useful even in cases that would be particularly challenging for existing program-repair techniques. In the future, we will extend our technique so that it can handle the more challenging case of faults involving multiple statements. Also, in order to improve MintHint's performance, we plan to investigate (1) alternative, more efficient techniques for building operational specifications and (2) outlier detection mechanisms that can further improving MintHint's tolerance to noise. Another interesting direction for future work is the investigation of how repair hints could be used to further automate the program-repair process. For example, we envision that hints could be used to inform program synthesis (*e.g.,* [16, 17, 22]) or sketching (*e.g.,* [37, 38]). A final future work direction is the study of the explanatory power of repair hints. We hypothesize that, unlike the fixes computed by fully automated repair techniques, repair hints can help developers better understand the nature of faults and corresponding repairs. It would be interesting to conduct investigations and experiments to assess the usefulness of repair hints in this direction.

7 ACKNOWLEDGEMENTS

We thank the volunteers who helped us in the user study. Sahana V.P. helped with MintHint's implementation during an internship at the Indian Institute of Science (IISc). The work by authors from IISc was partially supported by funding from the Robert Bosch Centre for Cyber-Physical Systems. The work by authors from Georgia Tech was partially supported by NSF awards CCF-1320783, CCF-1161821, and CCF-0964647, and by funding from Google, IBM Research and Microsoft Research.

8 REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *PRDC*, pages 39–46, 2006.

[2] A. Arcuri. On the Automation of Fixing Software Bugs. In *ICSE Companion*, pages 1003–1006, 2008.

[3] T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *POPL*, pages 97–105, 2003.

[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] V. Chandola, A. Banerjee, and V. Kumar. Outlier detection: A survey. *ACM Computing Surveys*, 2007.

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

[7] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[8] H. Cramér. *Mathematical Methods of Statistics*. Princeton Landmarks in Mathematics and Physics. Princeton University Press, 1945.

[9] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE*, pages 550–554, 2009.

[10] V. Debroy and W. E. Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *ICST*, pages 65–74, 2010.

[11] 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.

[12] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[13] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *ACM Sigplan Notices*, 2010.

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

[15] A. Groce, D. Kroening, and F. Lerda. Understanding Counterexamples with Explain. In *CAV*, pages 453–456, 2004.

[16] S. Gulwani. Automating String Processing in Spreadsheets using Input-Output Examples. In *PLDI*, pages 317–330, 2011.

[17] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing Geometry Constructions. In *PLDI*, pages 50–61, 2011.

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

[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and control flow-based test adequacy criteria. In *ICSE*, pages 191–200, 1994.

[20] T. Janssen, R. Abreu, and A. J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *ASE*, pages 662–664. IEEE Computer Society, 2009.

[21] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. Bugfix: A Learning-based Tool to assist Developers in Fixing Bugs. In *ICPC*, pages 70–79, 2009.

[22] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided Component-based Program Synthesis. In *ICSE*, pages 215–224, 2010.

[23] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated Concurrency-bug Fixing. In *OSDI*, pages 221–236, 2012.

[24] B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *CAV*, pages 287–294, 2005.

[25] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[26] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, pages 437–446, 2011.

[27] M. Kendall and J. Gibbons. *Rank correlation methods*. A Charles Griffin Book. E. Arnold, 1990.

[28] R. Konighofer and R. Bloem. Automated Error Localization and Correction for Imperative Programs. In *FMCAD*, pages 91–100, 2011.

[29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A Generic Method for Automatic Software Repair. *IEEE Trans. on Software Engineering*, pages 54 –72, 2012.

[30] F. Logozzo and T. Ball. Modular and Verified Automatic Program Repair. In *OOPSLA*, pages 133–146, 2012.

[31] H. D. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program Repair via Semantic Analysis. In *ICSE*, 2013.

[32] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Code-based Automated Program Fixing. In *ASE*, pages 392–395, 2011.

[33] 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.

[34] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[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] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. Alternate and learn: Finding witnesses without looking all over. In *CAV*, pages 599–615, 2012.

[37] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by Sketching for Bit-streaming Programs. In *PLDI*, pages 281–294, 2005.

[38] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[39] I. Vessey. Expertise in Debugging Computer Programs. *International Journal of Man-Machine Studies: A process analysis*, pages 459–494, 1985.

[40] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated Fixing of Programs with Contracts. In *ISSTA*, pages 61–72, 2010.

[41] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3):133–139, 1992.

[42] X. Zhang, N. Gupta, and R. Gupta. Locating Faults through Automated Predicate Switching. In *ICSE*, pages 272–281, 2006.

[43] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE 2006*, pages 272–281, 2006.