Synergistic Debug-Repair of Heap Manipulations

Sahil Verma
Department of Electrical Engineering,
Indian Institute of Technology Kanpur, India
vsahil@iitk.ac.in

Subhajit Roy
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, India
subhajit@cse.iitk.ac.in

ABSTRACT

We present Wolverine, an integrated Debug-Repair environment for heap manipulating programs. Wolverine facilitates stepping through a concrete program execution, provides visualizations of the abstract program states (as box-and-arrow diagrams) and integrates a novel, proof-directed repair algorithm to synthesize repair patches. To provide a seamless environment, Wolverine supports "hot-patching" of the generated repair patches, enabling the programmer to continue the debug session without requiring an abort-compile-debug cycle. We also propose new debug-repair possibilities, *specification refinement* and *specification slicing* made possible by Wolverine. We evaluate our framework on 1600 buggy programs (generated using fault injection) on a variety of data-structures like singly, doubly and circular linked-lists, Binary Search Trees, AVL trees, Red-Black trees and Splay trees; Wolverine could repair all the buggy instances within reasonable time (less than 5 sec in most cases). We also evaluate Wolverine on 247 (buggy) student submissions; Wolverine could repair more than 80% of programs where the student had made a reasonable attempt.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Formal software verification**; **Integrated and visual development environments**;

KEYWORDS

Program Debugging, Program Repair, Heap Manipulations

ACM Reference format:
Sahil Verma and Subhajit Roy. 2017. Synergistic Debug-Repair of Heap Manipulations. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106263

1 INTRODUCTION

We present Wolverine, an integrated debugging-and-repair tool for heap-manipulating programs. Wolverine hooks into gdb [2] to control the concrete execution of the buggy program, and extracts the concrete program state (via gdb) to provide visualizations of the abstract program states (as box-and-arrow diagrams). Such box-and-arrow diagrams are routinely used by programmers to plan heap manipulations and in online education [18].

Wolverine includes common debugging facilities like stepping through an execution, setting breakpoints, fast-forwarding to a breakpoint; we provide the important Wolverine commands in Table 1. Whenever the programmer detects an unexpected program state or control-flow (indicating a buggy execution), she can *repair* the box-and-arrow diagram to the expected state or force the expected control-flow (like forcing another execution of a `while` loop though the loop-exit condition is satisfied) during the debugging session. These expectations from programmer are captured by Wolverine as constraints to build a (partial) specification.

When the programmer feels that she has communicated enough constraints to the tool, she can issue a `repair` command, requesting Wolverine to attempt automated repair. Instead of aborting the current execution, and restarting it, Wolverine simulates *hot-patching* of the repair patch generated by its repair module, allowing the debugging session to continue from the same point. As the repair patch is guaranteed to have met all the user expectations till this point, the programmer can seamlessly continue the debugging session, with the repair-patch applied, without requiring an abort-compile-debug cycle.

Wolverine, by virtue of this seamless integration of debugging and repair, allows advanced debug-repair strategies wherein a skilled developer can communicate her domain knowledge of the program to Wolverine, thereby facilitating significant speedups during repair: if the programmer has confidence that a set of statements cannot have a bug, she can use *specification refinement* to eliminate these statements from the repair search space. Similarly, if the programmer understands that a part of the symbolic state is irrelevant for the current debug target, she can use *specification slicing* to eliminate them from the repair specification. Hence, rather than eliminating human expertise, Wolverine allows a synergistic human-machine interaction.

Wolverine bundles a novel *proof-directed repair strategy*: it generates a repair constraint that underapproximates the potential repair search space (via additional *underapproximation constraints*). If the repair constraint is satisfiable, a repair patch is generated. If a

Table 1: Wolverine cheatsheet

Command	Action
`start`	Starts execution
`enter`, `leave`	Enter/exit loop
`next`	Executes next statement
`step`	Step into a function
`change` e_s v_s	Set entity e_s to value v_s
`spec`	Add program state to specification
`repair`	Return repaired code
`rewrite`	Rewrite the patched file as a C program

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106263

```
1  struct node *head;
2  void reverse(){
3    struct node  *current, *temp1 = NULL, *temp2 = NULL;
4    current = head;
5    while (temp1 != NULL){    // FIX1: current != NULL
6      temp1 = current->prev;
7      temp2 = current->prev; // FIX2: current->next
8      current->prev = temp2;
9      current->next = temp1;
10     current = current->prev;
11   }
12   // head = temp1->prev;    // FIX3: insert stmt.
13 }
14 ...
15 int main(){
16   push(2);  push(4);  push(8); push(10);
17   reverse();
18 }
```

Figure 1: Our motivating example

proof of unsatisfiability is found (indicating a failed repair attempt) that does not depend on an underapproximation constraint, it indicates a buggy specification or a structural limitation in the tool's settings; else the respective underapproximation constraint that appears in the proof indicates the widening direction.

We evaluate Wolverine on a set of 1600 buggy files: 20 randomly generated faulty versions over four fault configurations of 20 benchmark programs collected from online sources [3] spanning multiple data-structures like singly, doubly and circular linked lists, Binary Search Trees, AVL trees, Red-Black trees and Splay trees. Wolverine could successfully repair all faults within reasonable time (less than 5 seconds for most of the programs).

We also evaluate Wolverine on 247 student submissions for 5 heap manipulating problems from an introductory programming course [10]; Wolverine could repair more than 80% of the programs where the student had made a reasonable attempt.

We make the following contributions in this paper:

- We propose that an integrated debug-repair environment can yield significant benefits; we demonstrate it by building a tool, Wolverine, to facilitate debug-repair on heap manipulations;
- We propose a new *proof-directed repair strategy* that uses the proof of unsatisfiability to guide the repair along the most promising direction;
- We propose advanced debugging techniques, *specification refinement* and *specification slicing*, that are facilitated by this integration of debugging and repair.

2 OVERVIEW

2.1 A Wolverine Debug-Repair Session

Let us demonstrate a typical debug-repair session on Wolverine: the program in Figure 1 attempts to create a doubly linked-list using the `push()` function, and then reverses it using the `reverse()` function. The `reverse()` function is buggy, with three faults:

(1) The loop condition is buggy which causes the loop to be traversed one less time than expected;
(2) The programmer (possibly due to a cut-and-paste error from the previous line) sets `temp2` to the `prev` field;
(3) Finally, she forgets to reset the `head` pointer to the new head of the reversed list.

The programmer launches Wolverine via the `start` command; then, she issues multiple `next` commands to concretely execute the `push()` statements, thereby creating the list. The current (symbolic) state of the program heap is displayed to the programmer (shown in Figure 2a).

```
(Wolverine) start
Starting program...
push(2)
(Wolverine) next; next; next; next;
```

```
push(4);
...
```

The user now decides to step into the `reverse()` function using the `step` command.

```
reverse();
(Wolverine) step
```

As the `reverse()` function progresses, `current` and `head` must be placed at desired places; on the other hand, `temp1` and `temp2` are used for intermittent manipulations of the pointers. Hence, the programmer uses the `track` command to intimate Wolverine of the expectations on `current` and `head` while allowing Wolverine to freely alter the temporary pointers for repairing the program.

```
struct node *current, *temp1 = NULL, *temp2 = NULL;
(Wolverine) track [current, head] [ ]
(Wolverine) next
```

```
current = head;
(Wolverine) next
```

When the `while` loop is reached, the user decides that the current program state is desirable at this point and, hence, asserts this state in the specification using the `spec` command. When invoked, the *repair module* will ensure that this program state is preserved (at this location) in any repair patch that it synthesizes.

```
while(temp1 != NULL)
(Wolverine) spec
Program states added
```

At the while statement, the execution would not enter the loop due to *Bug1*; however, as the user wants the execution to enter the loop, she alters the control-flow via the `enter` command to force the execution into the loop.

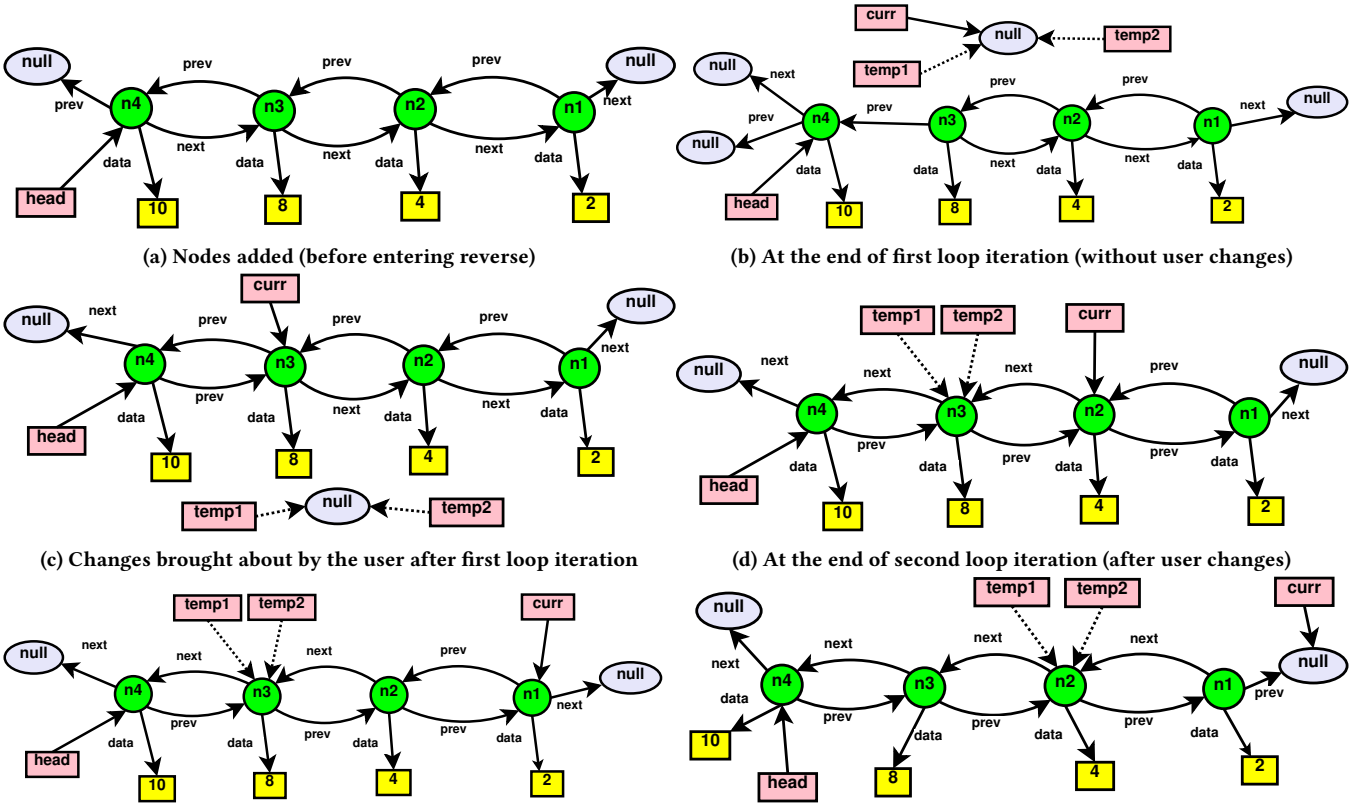
```
while(temp1 != NULL)
(Wolverine) enter
```

The user employs the `next` command to execute through the loop till the second iteration of the loop is hit.

```
temp1 = current->prev;
(Wolverine) next; next; next; next; next;
...
```

```
while(temp1 != NULL)
```

The state of the program that is displayed to the programmer is shown in Figure 2b: the user observes that the `prev` field of the node `n4`, which should have pointed to `n3`, instead points to `null`; the same is the case with the the pointer variable `current`. The user, thus, employs the `change` command in Wolverine to make the respective updates to the state.



(a) Nodes added (before entering reverse)

(b) At the end of first loop iteration (without user changes)

(c) Changes brought about by the user after first loop iteration

(d) At the end of second loop iteration (after user changes)

(e) At the end of third loop iteration (no changes required)

(f) At the end of fourth loop iteration

Figure 2: The visualization of the program execution provided by Wolverine. The dotted arrows denote relations in the program state that are `untracked`, i.e. not communicated to the repair module as part of the specification.

```
(Wolverine) change current n3
```

```
(Wolverine) change n4 -> prev n3
```

The state of the program heap is updated to as shown in Figure 2c. The user now confirms her expectations on the state of the program heap by issuing the `spec` command.

```
while(temp1 != NULL)
(Wolverine) spec
Program states added
```

The user now enters the loop for the second time.

```
(Wolverine) enter
while(temp1 != NULL)
. . .
```

At the end of this loop iteration, the user again finds undesirable changes in the linked list, so she changes the pointer `current` to `n2` and `prev` field of `n3` to `n2`.

```
while(temp1 != NULL)
(Wolverine) change current n2
(Wolverine) change n3 -> prev n2
```

The user, satisfied with the current (updated) state of the list (shown in Figure 2d), commits it to the specification.

```
while(temp1 != NULL)
(Wolverine) spec
Program states added
```

She now requests a repair patch using the `repair` command.

```
(Wolverine) repair
Repair synthesized...
```

The repair synthesized by Wolverine correctly fixes *Bug2*; however, the other bugs still remain as the trace does not contain these cases yet. The synthesized patch is guaranteed to satisfy the specification on the trace thus far; Wolverine, now, simulates *hot-patching* of this repair, allowing the user to continue this debugging session rather than having to undergo an abort-compile-debug cycle.

The user attempts to verify the repair by running the next (third) iteration of the loop; this iteration completes successfully with the expected heap state without the user making any changes (as shown in Figure 2e) alluding that the repair is possibly correct.

```
while(temp1 != NULL)
(Wolverine) enter
. . .
```

Fourth iteration also updates the program heap as per the expectations of the user, reinforcing her confidence in the repair patch.

```
while(temp1 != NULL)
(Wolverine) enter
. . .
```

Now, the user would like to exit the loop as the complete list has reversed; however, due to *Bug1*, the loop termination condition

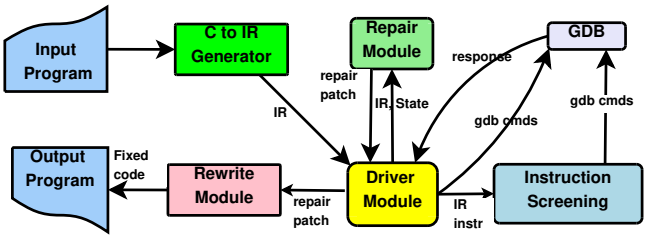


Figure 3: The claws of Wolverine

does not hold. The programmer forces a change in the control flow by using the `leave` command to forcefully exit the loop.

```
while(temp1 != NULL)
(Wolverine) leave
```

```
Exiting function...
```

Wolverine signals the user that it is returning from the function; at this point, the user recognizes that the state is still buggy as the `head` pointer continues to point to the node `n4` (rather than `n1`) as shown in Figure 2f. The user changes `head` to `n1`, adds the final states and requests a repair.

```
(Wolverine) change head n1
```

```
(Wolverine) spec
Program states added
```

```
(Wolverine) repair
Repair synthesized...
```

Note that this repair required the insertion of a new statement; Wolverine is capable of inserting a bounded number of additional statements to a buggy program. On our machine, the first repair call takes 1.5 s (fixing *Bug2*) while the second repair call returns in 6.2 s (fixing *Bug1* and *Bug3*).

To summarize, the programmer uses the debugging session to build a specification to drive automated repair. On the debugger, the programmer is required to "repair" the program states to indicate her expectations (specification) while the program is automatically repaired by Wolverine.

2.2 The Claws of Wolverine

Figure 3 shows the high-level schematic of Wolverine: the heart of Wolverine is the *driver* module that provides the user shell for accepting commands from the programmer and controls the coordination of the various modules.

Wolverine accepts the (buggy) C program from the user, and employs the *C-to-IR* generator to compile it into its intermediate representation (IR) as a sequence of guarded statements (Γ) and a location map (Λ) to map each line of the C-source code to an IR instruction (see §3). Note that each C-code instruction can get compiled down to multiple IR instructions and the same C-source line may contain multiple statements, each generating a sequence of IR instructions; for the sake of simplicity, we will assume the map Λ to be a one-to-one map in the rest of this paper (i.e. each C-source line appears in a different line and each C-statement compiles down to a single IR instruction).

On the programmer's command, the `driver` initiates the debug session by loading the binary on gdb: many of the command issued by the programmer are handled by dispatching a sequence of commands to gdb to accomplish the task. However, any progress of the program's execution (for example, the `next` command from the user) is routed via the *instruction screening* module that manages specification abstraction and simulates hot-patching (see §3 and Algorithm 2).

On the `repair` command, the *driver* invokes the *repair* module to request an automated repair based on the specification collected thus far. The repair module synthesizes a repair patch that is propagated to the *instruction screening* module to enable hot-patching of this repair. If the user is satisfied, she can invoke our *rewrite* module to translate the repaired program from our intermediate representation to a C-program.

3 ALGORITHM

We represent the state ($\mathcal{S}$) of a program that contains a set of variables σ_V and a set of heap nodes σ_H with fields σ_F as $\mathcal{V} \times \mathcal{H}$; the state of the program variables, $\mathcal{V}$, is a map $\sigma_V \rightarrow \mathcal{D}$ and the program heap is represented by $\mathcal{H}$ as a map $\sigma_H \times \sigma_F \rightarrow \mathcal{D}$. The domain of possible values, $\mathcal{D}$, is $\mathcal{I} \cup \sigma_H$ where $\mathcal{I}$ is the set of integers. For simplicity, we constrain the discussions in this paper to only two data-types: integers and pointers. We use the function $\Upsilon(e)$ to fetch the type of a program entity; a program entity $e \in \mathcal{E}$ is either a variable $v \in \sigma_V$ or a field of a heap node $h \in \sigma_H \times \sigma_F$. Also pointers can only point to heap nodes as we do not allow taking reference to variables.

We distinguish between *symbolic* and *concrete* states; the memory state witnessed by the concrete execution (via gdb) is referred to as the concrete state; we extract the symbolic state as a memory graph [41] from the concrete state, and then, essentially replace machine addresses by symbolic names. We maintain the heap nodes (pointers) in symbolic form while the scalar values (like integers) remain in their concrete form.

3.1 Symbolic Encoding of an Execution

Figure 4 shows the axiomatic semantics of our intermediate representation using Hoare triples [19]. Our intermediate representation maintains a program as a sequence of *guarded statements*: a statement is executed only if the corresponding guard evaluates to `true` (rules `grd1` and `grd2`). The scope of our repairs include the modification/insertion of both the statements and the guards.

The primary statements are the assignment statement ($x := y$), the getfield statement ($x := y.f$) and the putfield statement ($x.f = y$). For a map $\mathcal{Z}$, we use the notation $\mathcal{Z}_1 = \mathcal{Z}_2[e_1 \mapsto e_2]$ to denote that $\mathcal{Z}_2$ inherits all mappings from $\mathcal{Z}_1$ except that the mappings $e_1 \mapsto e_2$ is added/updated. We skip discussions of other statements in our IR (like `print`) for brievity.

The assignment statement requires a precondition that the types of the variables match; if the precondition holds, it updates the variable map $\mathcal{V}_1$ by adding a new mapping from the assigned variable `x` to the current value contained in `y`. The getfield statement requires type consistency and also that the dereferenced variable should be of pointer type and non-`null`. If the preconditions are met, the respective update ensues.

The `concrete` statement is used to enable an interesting debugging strategy (we refer to it as *specification refinement*): we invoke

$$\frac{\Upsilon(x) = \Upsilon(y) \quad \mathcal{V}_2 = \mathcal{V}_1[x \mapsto \mathcal{V}_1(y)]}{\{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ x := y\ \{\langle \mathcal{V}_2, \mathcal{H}_1 \rangle\}}\ asgn$$

$$\frac{\Upsilon(x) = \Upsilon(y.f) \quad \Upsilon(y) = \text{ptr} \quad \mathcal{V}_1(y) \neq \text{null} \quad \mathcal{V}_2 = \mathcal{V}_1[x \mapsto \mathcal{H}_1\langle \mathcal{V}_1(y), f \rangle]}{\{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ x := y.f\ \{\langle \mathcal{V}_2, \mathcal{H}_1 \rangle\}}\ getfld$$

$$\frac{\Upsilon(x) = \Upsilon(y.f) \quad \Upsilon(x) = \text{ptr} \quad \mathcal{V}_1(x) \neq \text{null} \quad \mathcal{H}_2 = \mathcal{H}_1[\langle \mathcal{V}_1(x), f \rangle \mapsto \mathcal{V}_1(y)]}{\{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ x.f := y\ \{\langle \mathcal{V}_1, \mathcal{H}_2 \rangle\}}\ putfld$$

$$\frac{\{\langle \overline{\mathcal{V}}_1, \overline{\mathcal{H}}_1 \rangle\}\ \zeta\ \{\langle \overline{\mathcal{V}}_2, \overline{\mathcal{H}}_2 \rangle\} \quad \mathcal{V}_2, \mathcal{H}_2 = Sym(\overline{\mathcal{V}}_2, \overline{\mathcal{H}}_2) \quad \overline{\mathcal{V}}_1, \overline{\mathcal{H}}_1 = Concr(\mathcal{V}_1, \mathcal{H}_1)}{\{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ concrete(\zeta)\ \{\langle \mathcal{V}_2, \mathcal{H}_1 \rangle\}}\ cncr$$

$$\frac{}{\{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ skip\ \{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}}\ skip$$

$$\frac{[\![grd]\!] = \text{true} \quad \{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ \text{stmt}\ \{\langle \mathcal{V}_2, \mathcal{H}_2 \rangle\}}{\{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ grd\ ?\ stmt\ \{\langle \mathcal{V}_2, \mathcal{H}_2 \rangle\}}\ grd1$$

$$\frac{[\![grd]\!] = \text{false}}{\{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}\ grd\ ?\ stmt\ \{\langle \mathcal{V}_1, \mathcal{H}_1 \rangle\}}\ grd2$$

Figure 4: Semantics of our intermediate representation

`concrete` with statement(s) ζ that we do not wish to model symbolically. We, in this case, extract a concrete precondition, execute ζ *concretely* on the precondition, fetching a concrete postcondition. The symbolic state corresponding to the concrete postcondition is assumed to be the postcondition of the `concrete` statement.

The guards are predicates that can involve comparisons from $\{\leq, <, \geq, >, =, \neq\}$ for integers and only $\{=, \neq\}$ for pointers; we skip providing their formal semantics.

When the `repair` command is invoked (say after executing `n` IR instructions via `gdb`), the *repair module* constructs the semantic model of the execution trace, Φ_{sem}, by the conjunction of the semantic encoding of the instructions in the trace:

$$\Phi_{sem} \equiv \prod_{i=1}^{n} \mathcal{T}_i(\mathcal{S}_i, \mathcal{S}_{i+1})$$

where $\mathcal{T}_i$ encodes the semantics of the i^{th} instruction (as shown in Figure 4) and $\mathcal{S}_i$ (and $\mathcal{S}_{i+1}$) denote the input (and output) state of this instruction (respectively).

3.2 The Heap Debugger

We show the basic functionalities provided by Wolverine in Algorithm 1. Our algorithm accepts a buggy program as a sequence of guarded statements $\langle \pi, \omega \rangle$ where π is a *guard* predicate of the form $\langle op, arg1, arg2 \rangle$, and ω is either an assignment ($v_1 = v_2$), a getfield operation ($v_1 = v_2.field$), a putfield operation ($v_1.field = v_2$) or a *concrete* statement.

Wolverine starts off with a string of initializations, where Θ is the set of "tracked" entities: the state of only these entities is recorded while creating the specifications. The algorithm, then, enters the command loop.

Algorithm 1: The Heap Debugger

```
1  Θ = {e | e ∈ H}
2  while true do
3      cmd := Prompt()
4      switch cmd do
5          case start do
6              loc = gdb_start()
7          case next do
8              loc = ExecuteStatement(loc)
9              S_c = fetch_concrete_state()
10             S_s, γ = create_symbolic_state(S_c, γ)
11             show(S_s)
12         case break <loc> do
13             gdb_send(break <loc>)
14         case track <[addlst]> <[remlst]> do
15             Θ := Θ ∪ addlst \ remlst
16         case change <e_s> <v_s> do
17             S_c = fetch_concrete_state()
18             S_s, γ = create_symbolic_state(S_c, γ)
19             S_s[e_s] := v_s
20             gdb_set_address(γ[e_s], v_s)
21         case spec do
22             S_c = fetch_concrete_state()
23             S_s, γ = create_symbolic_state(S_c, γ)
24             assert_spec(Λ[loc].IR_id, S_s ∩ Θ)
25         case repair do
26             patch := repair_run()
27             hot_patch(patch)
28     end
29 end
```

For the `next` command, Wolverine dispatches the next program statement to be executed (at source line `loc`) to the *statement screening module* (Algorithm 2), which returns the next line to be executed. Wolverine, then, queries `gdb` for the current program state (using the function `fetch_concrete_state()`), and then uses the function `create_symbolic_state()` to generate the memory map [41] and construct the symbolic state; this function returns back the symbolic map $\widehat{S}$ and a map γ; the map γ records the mapping of the symbolic entities to the concrete entities. The symbolic memory map is displayed to the user as a visual aid for debugging.

The `break` command dispatches the command to `gdb` for inserting a breakpoint. The `track` command, called with the list of entities to be added/removed from being tracked, updates the list.

The `change` command allows the user to alter the current program state (thereby alter the specification) by providing the symbolic entity e_s to be modified to the new value v_s. Wolverine accordingly, translates the symbolic state to a relevant concrete state and issues a string of `gdb` commands (summarized by the function `gdb_set_address()`) to alter the concrete program state.

The `spec` command asserts the symbolic state at the current program point, involving only the tracked entities.

Finally, the `repair` command invokes the *repair module* to synthesize a repair patch that obeys the string of assertions issued

Algorithm 2: ExecuteStatement

Input: $\Gamma :: [\langle stmt.action, stmt.grd, stmt.loc\rangle]$,
$\Lambda :: N \rightarrow \langle L, \{changed, inserted, preserved\}\rangle$

```
1  if stmt == "concrete" then
2  |  loc = gdb_send("next")
3  |  S_c = fetch_concrete_state()
4  |  S_s, γ = create_symbolic_state(S_c, γ)
5  |  repair_add_spec(Γ[pp].IR_id, S_s ∩ Θ)
6  else if Λ[loc].status == changed then
7  |  gdb_send("skip")
8  |  irstm = IR2gdbStm(Λ[loc].IR_id)
9  |  gdb_exec_stm(irstm)
10 else if Λ[loc].status == inserted then
11 |  irstm = IR2gdbStm(Λ[loc].IR_id)
12 |  gdb_exec_stm(irstm)
13 else  loc = gdb_send("next") ;
14 return loc
```

thus far. On a successful repair, it passes the repair patch to the *instruction screening* module to simulate hot-patching of the repair.

In addition to the above, Wolverine also supports altering of the control flow (like entry/exit of loops via the `enter` and `leave` commands respectively), flip branch directions etc. We demonstrated these features in §1 but we omit the details for brievity.

Algorithm 2 describes the `ExecuteStatement()` function implemented by the *instruction screening* module. This accepts a list of guarded statements Γ and a map Λ from the source line numbers (in N) to a tuple containing the corresponding IR instruction (in L) and status bits ($F \in \{changed, inserted, preserved\}$) to indicate: (a) the respective IR instruction has been modified (*changed*) by a repair patch, (b) appears as a new instruction (*inserted*) due to a repair patch, or (c) is unmodified (*preserved*); this information is required to simulate hot-patching. Note that deletion of a statement is also marked (*changed*) as the patch would simply set the guard to `false` in that case. The module handles two primary tasks:

- **Handling `concrete` statements** If a `concrete` statement is found, Wolverine executes the statement via gdb by issuing the `next` command. Then, it asserts the effect of this concrete execution by taking a snapshot of the concrete state (via gdb) and adding the corresponding symbolic state to the specification. This allows for a powerful debugging strategy; we refer to it as *specification refinement* (see §4).
- **Simulate hot-patching** If the statement has changed due to a repair patch, Wolverine requests gdb to skip the execution of the next statement. Then, it translates the "effect" of the modified statement into a sequence of gdb commands (*irstm*) via the `IR2gdbStm()` function and dispatches the command-list to gdb using our `gdb_exec_stm()` function.

Otherwise, it concretely executes the next statement via gdb by issuing the `next` command to it.

3.3 Proof-Guided Repair

Algorithm 3 shows our repair algorithm: it takes a (buggy) program $\mathcal{P}$ as a sequence of guarded statements and a bound on the number of new statements that a repair is allowed to insert

Algorithm 3: Unsat Core Guided Repair Algorithm

```
1  Φ_grd = Φ_stm = Φ_ins = ∅
2  n := |P| + num_insert_slots
   /* Assert the input (buggy) program                    */
3  for i ∈ {1 ... |P|} do
4  |  Φ_grd += ⟨¬r_ξ(i) ⟹ (P̂.grd[ξ(i)] == P.grd[i])⟩
5  |  Φ_stm += ⟨¬s_ξ(i) ⟹ (P̂.stm[ξ(i)] == P.stm[i])⟩
6  end
   /* Initialize the insertion slots                      */
7  for i ∈ {|P| ... n} do
8  |  Φ_ins += ⟨¬t_ξ(i) ⟹ (φ_ξ(i) == false)⟩
9  end
   /* Define the placing function ξ                       */
10 Φ_bk := ∀_{i∈{1...n}}(1 ≤ ξ(i) ≤ n) ∧ distinct(ξ(i))
11 Φ_bk += ∀_{⟨i,.,.⟩,⟨k,.,.⟩∈P}(i < k ⟹ ξ(i) < ξ(k))
12 v := UNSAT
   /* Relax till specification is satisfied               */
13 τ_grd := τ_stm := τ_ins := 0
14 while v = UNSAT or tries exceeded do
15 |  ⟨res, P̂, uc⟩ := Solve(Φ_spec ∧ Φ_sem ∧ Φ_bk,
16 |                        Φ_grd ∧ Σ_{k∈{1...|P|}} r_k < τ_grd,
17 |                        Φ_stm ∧ Σ_{k∈{1...|P|}} s_k < τ_stm
18 |                        Φ_ins ∧ Σ_{k∈{|P|+1...n}} t_k < τ_ins)
   |  /* Use the UNSAT core to drive relaxation          */
19 |  if res = UNSAT then
20 |  |  if Φ_grd ∩ uc ≠ ∅ then  τ_grd += 1 ;
21 |  |  else if Φ_stm ∩ uc ≠ ∅ then  τ_stm += 1 ;
22 |  |  else if Φ_ins ∩ uc ≠ ∅ then  τ_ins += 1 ;
23 |  |  else  return null ;
24 end
25 if tries exceeded then  return null ;
26 return P̂
```

(`num_insert_slots`). The repair algorithm attempts to search for a repair candidate $\widehat{\mathcal{P}}$ (of size $n = |\mathcal{P}|$ + `num_insert_slots`) that is "close" to the existing program and satisfies the programmer's expectations (specification). Our algorithm is allowed to mutate and delete existing statements, and insert at most n new statements. The insertion slots contain a guard `false` to begin with (Line 8); the repair algorithm is allowed to change it to "activate" the statement. Deletion of a statement changes the guard of the statement to `false`.

Wolverine allows for new nodes and temporary variables by providing a bounded number of additional (hidden) nodes/temporaries, made available on demand. The number of insertion slots is configured by the user, but these slots are activated by the repair algorithm only if needed. For loops, we add additional constraints so that all loop iterations encounter the same instructions.

3.3.1 Primary Constraints. We use a set of selector variables $\{r_1, \ldots, r_n, s_1, \ldots, s_n\}$ to enable a repair. Setting a selector variable to `true` relaxes the respective statement, allowing Wolverine to synthesize a new guard/statement at that program point to satisfy

the specification. We define a metric, $closeness(\mathcal{P}_1, \mathcal{P}_2)$, to quantify the distance between two programs by summing up the set of guards and statements that match at the respective lines. As the insertion slots should be allowed to be inserted at any point in the program, the closeness metric would have to be 'adjusted' to incorporate this aberration due to insertions. For this purpose, our repair algorithm also infers a relation ξ that maps the instruction labels in the repair candidate $\widehat{\mathcal{P}}$ to the instruction labels in the original program $\mathcal{P}$; the instruction slots are assigned labels from the set $\{|\mathcal{P}|+1, \ldots, n\}$. We define our closeness metric as:

$$\begin{aligned}\texttt{closeness}(\mathcal{P}, \widehat{\mathcal{P}}) = &\sum_{i=1}^{|\mathcal{P}|} (\mathcal{P}.grd[i] = \widehat{\mathcal{P}}.grd[\xi(i)]) \\ &+ \sum_{i=1}^{|\mathcal{P}|} (\mathcal{P}.stm[i] = \widehat{\mathcal{P}}.stm[\xi(i)]) \\ &+ \sum_{i=|\mathcal{P}|+1}^{n} (\widehat{\mathcal{P}}.grd[\xi(i)] \neq \texttt{false})\end{aligned}$$

The above metric weights a repair candidate by the changes in the statements/guards and new statements added (insertion slots *activated*).

Algorithm 3 starts off by asserting the input program $\mathcal{P}$, via the selector variables, as part of the constraints Φ_{grd} and Φ_{stm} (lines 3–6), and initializes the insertion slots to their deactivated state (lines 7–9) with selector variables t_i. The constraint Φ_{bk} ensures that the function ξ is well-formed: for each i, $\xi(i)$ is a distinct value in the range $\{1 \ldots n\}$ and is a monotonically increasing function (this ensures that the statements preserve the same order in $\widehat{\mathcal{P}}$ as the order in $\mathcal{P}$).

Finally, it uses issues a Solve() query to an SMT solver to solve the repair constraint; the sub-constraint Φ_{sem} contains the semantic encoding of our intermediate statements (Figure 4) and Φ_{spec} contains the specification collected during the debugging session as a result of the spec commands.

3.3.2 Proof-Guided Search Space Widening. To ensure that the repaired candidate program $\widehat{\mathcal{P}}$ is *close* to the original program, we progressively relax the closeness bounds. The variables τ_{grd}, τ_{stm}, and τ_{ins} constrain the distance (in terms of changed guards, statements and activated insertion slots) of a repair candidate from the original program.

We use a **Proof-Guided Repair Strategy**: we use the unsat core (*uc*) produced from the proof of unsatisfiability to direct us to the bound that needs to be relaxed. The unsat core represents the central *reason* as to why the program cannot be made to satisfy the specification; if a constraint $\langle s_i \implies \ldots \rangle$ is found in the unsat core, it implies that the reason for unsatisfiability *may* be attributed to the fact that s_i is false! Hence, one possible way to remove this unsatisfiability is to increase the bound on τ_{stm} that allows s_i to turn `false`.

At the same time, we would also like to enforce a priority on the relaxations; for instance, deletion of a statement or mutation of a guard can be considered "smaller" changes than changing a statement, or worse, inserting a new statement. The chain of conditions (lines 20–22) ensures that, if the unsat core directs us to a possibility of smaller change, we relax the respective bound before others. Finally, on a successful repair, we return the repaired program $\widehat{\mathcal{P}}$.

Guiding repair via the unsat proof has multiple advantages:

- The unsatisfiability core (uc) guides us to a feasible repair; for example, if *uc* does not contain the constraints pertaining to activation of the insertion slots, then it is unlikely that inserting a new statement will fix the bug;
- It allows us to prioritize the repair actions; one would prefer mutation of a statement than insertion of a new statement;
- The strategy is fast as the solver is provided constrained search spaces, which is incrementally widened (in a direction dictated by the proofs) as the search progresses. In case the program to be repaired is close to the original program, the solver will be provided only "easy" instances that are allowed to mutate/insert a small number of statements;
- It allows a fail-fast (line 23) if the specification is buggy or the repair is not possible due to structural constraints (like the number of insertion slots provided); if *uc* does not contain any constraint from $\{\Phi_{grd}, \Phi_{stm}, \Phi_{ins}\}$, then the program cannot be repaired via any repair action without violating the hard constraints (like the program semantics).

The unsat core not only identifies the possible culprits (a sort of bug localization) but also allows us to define a priority among our repair preferences. To the best of our knowledge, ours is the first repair algorithm that uses unsat proofs to direct repair; however, this idea has threads of similarity with a model-checking algorithm, referred to as underapproximation widening[17] (see §6).

We evaluated a variant (**AlgVar**) of our proof directed repair scheme: instead of increasing the respective repair bound, we randomly relax one of the constraints from the unsat core. However, we found that the unsat cores are poor—quite far from the minimum unsat core. Hence, this variant of our algorithm performs poorly, both in terms of success-rate and the time taken for repair (see §5).

4 ADVANCED DEBUGGING/REPAIR

4.1 Specification Refinement

Wolverine is designed to model heap manipulations; however, Wolverine can use the `concrete(ζ)` statement in its intermediate representation as an abstraction of any statement ζ that it does not model. On hitting a `concrete(ζ)` statement, Wolverine uses `gdb` to concretely execute the statement and updates its symbolic state from the concrete states provided by `gdb`. Figure 6 shows an instance where we wrap the `i=i+1` statement in an concrete execution; Wolverine translates this statement to a string of `gdb` commands, and the symbolic state is updated with the value of i from the concrete state that `gdb` returns after executing the statement. Hence, though Wolverine is specifically targeted at heap manipulations, it can be used to debug/repair programs containing other constructs as well as long as the bug is in a heap manipulation. We refer to this technique of reconstructing the symbolic specification by running the statement concretely as *specification refinement*.

Specification refinement can be used in creative ways by skilled engineers. In Figure 5, the programmer decided to wrap a complete function call (`foo()`) within the `concrete()` construct, allowing Wolverine to reconstruct the effect of the function call via concrete execution without having to model it. This strategy can fetch

```
1 void bar(){
2   struct node *current= NULL; int i;
3   current = head;
4   while (current != NULL){
5     concrete[i = foo();]       // concrete  stmt.
6     current->data = i;
7     current = current->next;
8   }
9 }
```

Figure 5: Refinement with concrete function calls

```
1 void reverse(int i){
2   struct node * last, *current, *nt = NULL;
3   current = head;
4   while (current != NULL){
5     nt = current->next;
6     current->next = prev;
7     prev = current;
8     current->data = j;  // FIX1: current->data = i
9     concrete[i = i+1;]  // concrete stmt.
10    current = nt;       // FIX2:  current = prev;
11  }
12  // head = prev;       // FIX3:  insert  stmt.
13 }
```

Figure 6: Example for specification slicing

significant speed-ups for repair: let us assume that, in Figure 1, the programmer uses her domain knowledge to localizes the fault to lines 6–8; she can pass this information to Wolverine by wrapping the other statements in the loop (lines 5,9) in `concrete` statements; this hint brings down the repair time on the full program on the complete execution from 6.0s to 1.5s, i.e achieving a 4× speedup (on our machine). This is understandable as each instruction that is modeled can increase the search space exponentially.

4.2 Specification Slicing

In case the programmer is aware that two features of her implementation can be debugged independently, she can employ another interesting debugging feature, that we refer to as *specification slicing*: she can use the `track` command in Wolverine to *slice away* the values that are irrelevant to the feature being debugged. For example, Figure 6 shows a program that reverses a singly linked list while also assigning the `data` field with value increasing by one. There exist three bugs: the bugs at line 10 and 12 affects the loop reversal while the bug at line 8 controls the assignment to `data`. The programmer, can employ the `track` command to remove the `data` field from the specifications for faster repair of *Bug2* and *Bug3* (i.e. the blue edges in Figure 7); subsequently, for *Bug1* the red edges are dropped. The red and blue edges in Figure 7 shows how one of the states can be sliced into multiple (smaller) states for repair. On our machine, the two slices took 3.0s (*Fix2* and *Fix3*) and 1.5s (*Fix1*) to be fixed while the complete specification (without slicing) took 6.5s.

5 EXPERIMENTS

We built Wolverine using the `gdb` Python bindings[1]; the C-to-AST compiler uses `pycparser`[5]; the visualization module uses `igraph`[4] to construct the box and arrow diagrams and the repair module uses the Z3[6] theorem prover to solve the SMT constraints. We conduct our experiments an Intel(R) Xeon(R) CPU @ 2.00GHz

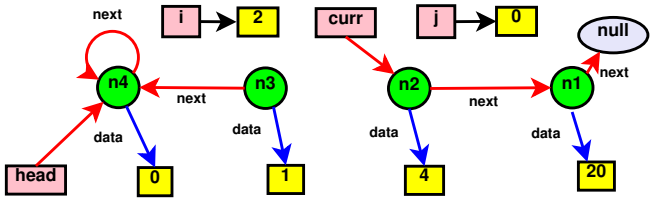


Figure 7: Specification slicing

machine with 32 GB RAM. To evaluate our implementation, we attempt to answer the following research questions:

- **RQ1** Is our repair algorithm able to fix different types and combination of bugs in a variety of data-structures?
- **RQ2** Can our repair algorithm fix these bugs in reasonable time?
- **RQ3** How does our repair algorithm scale as the number of bugs are increased?
- **RQ4** Is Wolverine capable of debugging/fixing real bugs?

5.1 Experiments with Fault-Injection

For RQ1 and RQ2, we select 20 heap manipulating programs (Table 2) from online sources[3] for a variety of data-structures. To create buggy versions, we build our own fault injection engine to automatically inject bugs (at random) thereby eliminating possibilities of human bias. For each program, we control our fault-injection engine to introduce a given number of bugs; we characterize a buggy version by $\langle x, y \rangle$ implying that the program requires modification of x (randomly selected) program expressions (each modification is a replacement of a program variable or a field in a program statement or a guard) and the insertion of y newly synthesized program statements.

For the experiments, Wolverine makes 10 attempts at repairing a program, each attempt followed by *proof-directed search space widening*; each attempt is run with a timeout of 30s. The experiment was conducted in the following manner:

(1) We evaluate each benchmark (in Table 2) for four bug classes: Class1 ($\langle 1, 0 \rangle$), Class2($\langle 1, 1 \rangle$), Class3($\langle 2, 0 \rangle$) and Class4($\langle 2, 1 \rangle$);
(2) For each benchmark B_i, at each bug configuration $\langle x, y \rangle$, we run our fault injection engine to create *20 buggy versions* with x errors that require modification of an IR instruction and y errors that require insertion of a new statement;
(3) Each of the above buggy program is run 2 times to amortize the run time variability.

Figure 4 shows the average time taken to repair a buggy configuration over the 20 buggy variants, which were themselves run twice (the reported time shows the average time taken for the successful repairs only). We report the time taken for our main algorithm (in Algorithm 3) and its variant **AlgVar** (discussed in the last paragraph in §3). Our primary algorithm performs quite well, fixing most of the repair instances in less than 5 seconds; understandably, the bug classes that require insertion of new instructions (Classes 2 and 4) take longer. There were about 1–4 widenings for bugs in class 1,2,3; the bugs in class 4 were more challenging needing 2–6 widenings.

In terms of the success rate, *our primary algorithm was able to repair all the buggy instances.* However, Figure 8c shows the success rate for each bug configuration for **AlgVar**; the success rate is computed as the fraction of buggy instances (of the given buggy

Table 2: Description of our benchmarks[3]

B1	Reverse singly linked-list	B2	Reverse doubly linked-list	B3	Deletion from singly linked-list
B4	Creation of circular linked-list	B5	Sorted Insertion singly linked list	B6	Insertion in single linked list
B7	Swapping nodes singly linked list	B8	Splaytree Left Rotation	B9	Find minimum in Binary Search Tree
B10	Find Length of singly linked list	B11	Print all nodes singly linked list	B12	Splitting of circular linked list
B13	AVL tree right rotation	B14	AVL tree left-right rotation	B15	AVL tree left rotation
B16	AVL tree right-left rotation	B17	Red-Black tree left rotate	B18	Red-Black tree right rotate
B19	Enqueue using linked-list	B20	Splaytree Right Rotated		

Table 3: Tool evaluation on student submissions

Id	Total	Fixed	ImLmt	OoScope	Vacuous
S1	47	30	2	8	7
S2	48	29	3	8	8
S3	48	36	0	5	7
S4	61	46	0	6	9
S5	43	25	0	4	14

configuration) that could be repaired by Wolverine (in any of the two attempts).

The inferior performance of the variant of our main algorithm shows that the quality of the unsat cores is generally poor, while the performance of our primary algorithm demonstrates that even these unsat cores can be used creatively to design a good algorithm.

Figure 9 answers RQ3 by showing the scalability of Wolverine with respect to the number of bugs on 5 of our benchmarks. We see that in most of the benchmarks, the time taken for repair grows somewhat linearly with the number of bugs though (in theory) the search space grows exponentially. Also, one can see that more complex manipulations like left-rotation in a red-black tree (B17) are affected more as a larger number of bugs are introduced as compared to simpler manipulations like inserting a node in a sorted linked list (B5).

The variance in the runtimes for the different buggy versions, even for those corresponding to the same buggy configuration, was found to be high. This is understandable as SMT solvers often find some instances much easier to solve than others even when the size of the respective constraint systems is similar.

5.2 Experiments with Student Submissions

In order to answer RQ4, we collected 247 buggy submissions from students corresponding to 5 programming problems on heap manipulations from an introductory programming course [10]. We attempted repairing these submissions and categorized a submission into one of the following categories:

- Fixed (**Fixed**) These are the cases where Wolverine could automatically fix the errors.
- ImLmt (**Implementation Limitations**) These are cases where, though our algorithm supports these repairs, the current state of our implementation could not support automatic repair.
- OoScope (**Out of scope**) The bug in the submission did not occur in a heap-manipulating statement.
- Vacuous (**Vacuous**) In these submissions the student had hardly attempted the problem (i.e. the solution is almost empty).

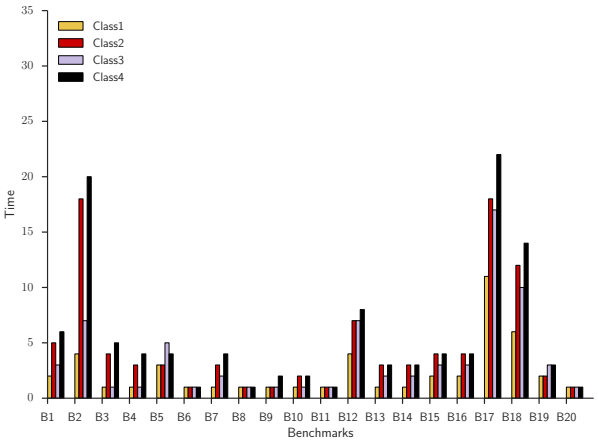
Overall, we could repair more than 80% of the submissions automatically where the student has made some attempt at the problem (i.e. barring the vacuous cases).

6 RELATED WORK

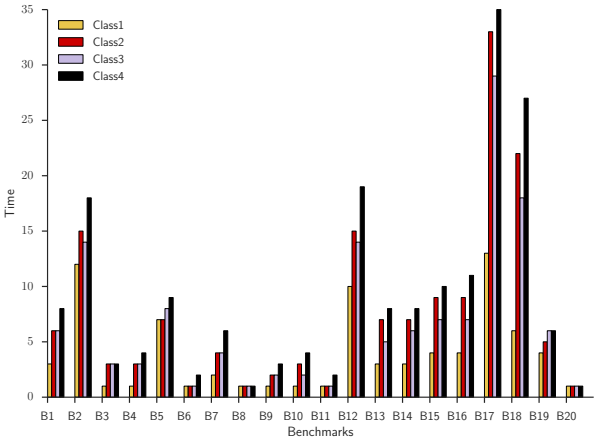
Our proof guided repair algorithm is inspired by a model-checking technique for concurrent programs—referred to as underapproximation widening [17], that builds an underapproximate model of the program being verified by only allowing a certain set of thread interleavings by adding an *underapproximation constraint* that inhibit all others. If the verification instance finds a counterexample, a bug is found. If a proof is found which does not rely on the underapproximation constraint, the program is verified; else, it is an indication to relax the underapproximation constraint by allowing some more interleavings. Hence, the algorithm can find a proof from underapproximate models without needing to create abstractions. To the best of our knowledge, ours is the first attempt at adapting this idea for repair. In the case of repairs, performing a proof-guided search allows us to work on smaller underapproximated search spaces that are widened on demand, guided by the proof; at the same time, it allows us to prioritize among multiple repair strategies like insertion, deletion and mutation. In the space of repairs, DirectFix [29] also builds a semantic model of a program but instead uses a MAXSAT solver to search for a repair. Invoking a MAXSAT solver is not only expensive, a MAXSAT solver also does not allow prioritization among repair strategies. In DirectFix, it is not a problem as the tool only allows mutation of a statement for repair and does not insert new statements. Alternatively, one can use a weighted MAXSAT solver for the prioritizing among repair actions, but it is prohibitively expensive; we are not aware of any repair algorithm that uses a weighted MAXSAT solver for repair.

Zimmermann and Zeller [41] introduce *memory graphs* to visualize the state of a running program, and Zeller used memory graphs in his popular *Delta Debugging* algorithms [39, 40] to localize faults. Our algorithm is also based on extracting these memory graphs from a concrete execution on gdb and employing its symbolic form for repair. The notion of `concrete` statement in Wolverine bears resemblance to concolic testing tools [15, 35].

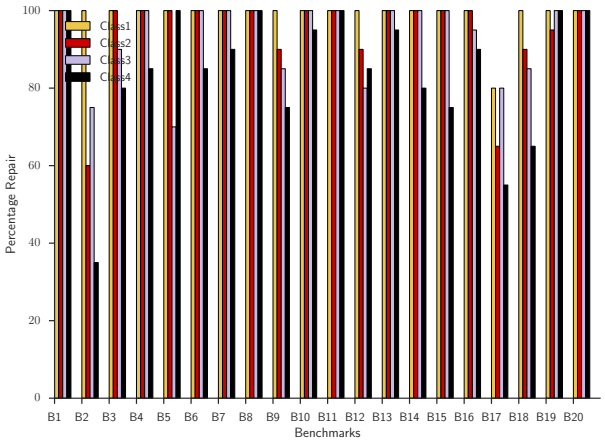
Symbolic techniques [7, 16, 21, 22, 27] build a symbolic model of a program and use a model-checker to "execute" the program; they classify a statement buggy based on the "distances" of faulty executions from the successful ones. Angelic Debugging [9], instead, uses a symbolic execution engine for fault localization by exploring alternate executions on a set of suspicious locations, while Angelix [30, 32] fuses angelic debugging-style fault localization with a component-based synthesis [20] framework to automatically synthesize fixes. There have also been regression aware strategies to localize/repair bugs [8]. There have also been proposals to use statistical techniques [25, 26, 31], evolutionary search [24, 33, 37, 38] and probabilistic models [28] for program debugging. However, the above algorithms, through quite effective for arithmetic programs,



(a) Repair time for our primary algorithm



(b) Repair time for the variant (AlgVar) of our primary algorithm



(c) Success rates for AlgVar; our primary algorithm has 100% success rate in all cases.

Figure 8: Performance of our repair algorithms

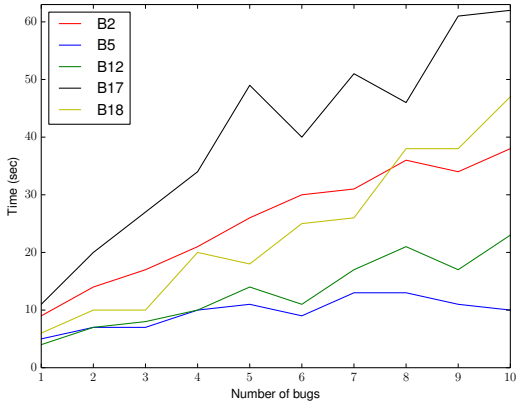


Figure 9: Experiment showing the increase in repair time with increasing number of bugs

were not designed for debugging/repairing heap manipulations. There have been proposals that repair the state of a data-structure on-the-fly whenever any consistency check (from a set of checks provided by a user) is found to fail [11, 12]. However, our work is directed towards fixing the bug in the source code rather than in the state of the program, which makes this direction of solutions completely unrelated to our problem. In the space of functional programs, there has been a proposal [13, 23] to repair functional programs with unbounded data-types; however, such techniques are not applicable for debugging imperative programs.

There has been some work in the space of synthesizing heap manipulations. The storyboard programming tool [36] uses abstract specifications provided by the user in three-valued logic to synthesize heap manipulations. As many users are averse to writing formal specification, SYNBAD [34] allows synthesis of programs from concrete examples; to amplify the user's confidence in the program, it also includes a test-generation strategy on the synthesized program to guide refinement. The intermediate representation of Wolverine is inspired from SYNBAD; Wolverine can also be extended with a test-generation strategy to validate the repair on a few more tests before exposing it to the programmer. SYNLIP [14] proposes a linear programming based synthesis strategy for heap manipulations. Feser et al. [13] proposes techniques for synthesis of functional programs over recursive data structures. Wolverine, on the other hand, attempts repairs; the primary difference between synthesis and repair is that, for a "good" repair, the tool must ensure that the suggested repair only makes "small" changes to the input program rather than providing a completely alternate solution.

7 DISCUSSION

We believe that tighter integration of dynamic analysis (possibly enabled by a debugger) and static analysis (via symbolic techniques) can open new avenues for debugging tools. We were careful to select a variety of data-structures and injected bugs via an automated fault injection engine to eliminate human bias; nevertheless, more extensive experiments can be conducted.

REFERENCES

[1] GDB Python API. https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html. Online; accessed 24 January 2017.

[2] GDB: The GNU Project Debugger. https://sourceware.org/gdb/. Online; accessed 24 January 2017.

[3] Geeks for geeks. http://www.geeksforgeeks.org/data-structures/. Online; accessed 24 January 2017.

[4] igraph – the network analysis package. http://igraph.org/python/. Online; accessed 24 January 2017.

[5] Pycparser: C parser in Python. https://pypi.python.org/pypi/pycparser. Online; accessed 24 January 2017.

[6] The Z3 Theorem Prover. https://github.com/Z3Prover/z3/wiki. Online; accessed 24 January 2017.

[7] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 97–105, New York, NY, USA, 2003. ACM.

[8] Rohan Bavishi, Awanish Pandey, and Subhajit Roy. To be precise: Regression aware debugging. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 897–915, New York, NY, USA, 2016. ACM.

[9] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.

[10] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani. Prutor: A system for tutoring CS1 and collecting student programs for analysis. *CoRR*, abs/1608.03828, 2016.

[11] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 78–95, New York, NY, USA, 2003. ACM.

[12] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 855–858, New York, NY, USA, 2008. ACM.

[13] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 229–239, New York, NY, USA, 2015. ACM.

[14] Anshul Garg and Subhajit Roy. Synthesizing heap manipulations via integer linear programming. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, pages 109–127. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[16] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error Explanation with Distance Metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, June 2006.

[17] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 122–131, New York, NY, USA, 2005. ACM.

[18] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[20] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[21] Manu Jose and Rupak Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 504–509, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Manu Jose and Rupak Majumdar. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

[23] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 217–233. Springer International Publishing, Cham, 2015.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, January 2012.

[25] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[26] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: Statistical Model-based Bug Localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 286–295, New York, NY, USA, 2005. ACM.

[27] Yongmei Liu and Bing Li. Automated Program Debugging via Multiple Predicate Switching. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 327–332. AAAI Press, 2010.

[28] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[29] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015.

[30] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[31] Varun Modi, Subhajit Roy, and Sanjeev K. Aggarwal. Exploring Program Phases for Statistical Bug Localization. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 33–40, New York, NY, USA, 2013. ACM.

[32] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[33] ThanhVu Nguyen, Westley Weimer, Claire Le Goues, and Stephanie Forrest. Using execution paths to evolve software patches. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 152–153. IEEE, 2009.

[34] Subhajit Roy. From concrete examples to heap manipulating programs. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 126–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[35] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[36] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 289–299, New York, NY, USA, 2011. ACM.

[37] Westley Weimer. Patches As Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 181–190, New York, NY, USA, 2006. ACM.

[38] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[39] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

[40] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

[41] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, pages 191–204, London, UK, UK, 2002. Springer-Verlag.