MIMIC: Locating and Understanding Bugs by Analyzing Mimicked Executions

Daniele Zuddas*, Wei Jin†, Fabrizio Pastore*
Leonardo Mariani*, Alessandro Orso†

*University of Milano - Bicocca
Milano, ITALY
{zuddas | pastore | mariani}@disco.unimib.it

†Georgia Institute of Technology
Atlanta, GA, USA
{weijin | orso}@gatech.edu

ABSTRACT

Automated debugging techniques aim to help developers locate and understand the cause of a failure, an extremely challenging yet fundamental task. Most state-of-the-art approaches suffer from two problems: they require a large number of passing and failing tests and report possible faulty code with no explanation. To mitigate these issues, we present MIMIC, a novel automated debugging technique that combines and extends our previous input generation and anomaly detection techniques. MIMIC (1) synthesizes multiple passing and failing executions similar to an observed failure and (2) uses these executions to detect anomalies in behavior that may explain the failure. We evaluated MIMIC on six failures of real-world programs with promising results: for five of these failures, MIMIC identified their root causes while producing a limited number of false positives. Most importantly, the anomalies identified by MIMIC provided information that may help developers understand (and ultimately eliminate) such root causes.

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

General Terms: Reliability, Experimentation

Keywords: Debugging, execution synthesis, anomaly detection

1. INTRODUCTION

Because software debugging is an extremely expensive and human intensive activity, researchers and practitioners alike have put a great deal of effort into developing automated debugging techniques and tools that can help developers reduce the cost of debugging. One debugging task in particular, fault localization, has been intensively investigated during the last decade. Among the fault localization techniques developed to date, the ones based on statistical analysis are particularly popular (*e.g.,* [2, 17, 19, 21]).

Although statistical-fault-localization techniques can be effective in guiding the developer towards parts of the code that are likely to be responsible for an observed failure, most of these approaches have two serious limitations. First, in order to perform their statistical analysis, they require a large number of *suitable* passing and failing test cases [3, 32]. Unfortunately, this ideal set of tests is rarely available in practice. Second, these approaches normally return a list of statements ranked according to their likelihood of being faulty, with no additional information. As the study by Parnin and Orso has shown [24], developers usually need context and some form of explanation to understand a bug.

To address and mitigate these issues, we present MIMIC, a novel automated debugging technique that combines and extends two techniques developed in previous work by the authors. The first one is an input generation technique that can synthesize executions similar to an observed failure [16]. The second technique performs anomaly detection by identifying the violations of a previously built model of the normal behavior of an application [26]. Given a program p and a failure f for p, MIMIC performs the following steps. First, it generates a set of inputs that, when run against p, result in both passing and failing executions that are "similar" to f. Second, MIMIC leverages these synthesized executions and the structure of the program to automatically compute suitable points where to monitor program behavior. Third, MIMIC uses the passing inputs to build a model of p's normal behavior, and the failing inputs to identify violations of this model at the monitoring points computed in the previous step. Finally, MIMIC reports the discovered violations, suitably processed, as potential explanations for f.

To evaluate MIMIC, we implemented it in a prototype tool and used the tool on six failures of real-world programs. The results of our evaluation provide initial evidence of the usefulness of our approach. In particular, the results show that MIMIC can report anomalies that are closely related to the root cause(s) of f and can potentially help understand such cause(s). In addition, the results show that both (1) using the synthesized executions generated by MIMIC, instead of p's existing test cases, and (2) monitoring program behavior at the program points selected by MIMIC, rather than at arbitrary program points, can considerably improve the effectiveness of the approach.

The main contributions of this work are:

- The definition of MIMIC, a new highly automated technique for discovering failure causes by leveraging only failure data from a single failing execution.
- An approach for identifying effective monitoring points within a program for anomaly detection.
- An optimization heuristic for filtering out spurious anomalies that are unlikely to be related to failure causes.
- The development of a publicly available prototype tool that implements our technique, available at `http://www.lta.disco.unimib.it/tools/mimic`.
- An empirical study that shows initial, yet clear evidence of the usefulness of our approach.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2643014.

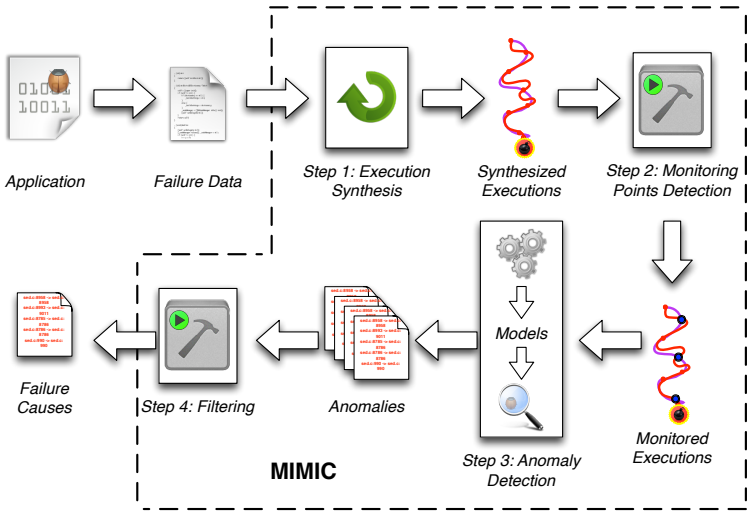

Figure 1: Overview of the MIMIC approach.

The rest of the paper is organized as follows. Section 2 provides background information that is necessary to make the paper self contained. Section 3 describes our approach in detail. Section 4 presents our empirical evaluation. Finally, Sections 5 and 6 discuss related work and conclude the paper.

2. BACKGROUND

Before discussing MIMIC in detail, we concisely describe the techniques that we leverage and extend in this work: F^3 and RADAR.

2.1 F^3

F^3 is a general approach for reproducing and debugging field failures that we developed in previous work [16]. The inputs to F^3 are a program P and failure data D for an execution e of P that resulted in a field failure f. Given this input, F^3 (1) synthesizes multiple passing and failing executions "similar" to e and (2) applies a customized statistical fault localization approach to identify and rank statements in P that are likely to be responsible for f.

The failure data used by F^3, D, is an ordered list of intermediate program locations that can be seen as breadcrumbs to be followed, or subgoals to be reached, to get to the failure. The last goal in the list is the actual failure point. Given D, F^3 uses our BugRedux algorithm [15] to perform an optimized guided forward symbolic execution that aims to synthesize multiple passing and failing executions of P that reach the goals in the same order as e. (To generate passing execution, the algorithm may remove some intermediate goals from D, which increases the degrees of freedom of the synthesis, and thus, the chances of generating such executions.)

In MIMIC, we leverage F^3 in two ways. First, we use the execution generator in F^3 to generate multiple passing and failing executions similar to the observed (in-house, in this case) failure f. As failure data, MIMIC uses a (partial) call sequence for f, which in previous work we found to offer a good tradeoff between amount of information collected and usefulness of the information in guiding execution synthesis [15] and fault localization [16]. Second, MIMIC leverages the suspiciousness values generated by F^3's customized fault localization techniques to select suitable points where to monitor program behavior for anomaly detection.

2.2 RADAR

RADAR is a technique for automatically identifying the likely causes of a regression failure [26]. RADAR relies on the availability of a regression test suite such that (1) all tests in the suite pass when executed on the original version of the software, and (2) some tests fail when executed on the modified software. The basic idea behind RADAR is to identify the behavioral differences between the failing executions of the modified program and the passing executions of the original program.

The behavior of the original and modified programs is observed at a set of code locations determined according to the differences between the two programs. When running the programs, RADAR collects two pieces of dynamic information: the sequence of statements executed and the values of the program variables in scope at the monitoring points. The information collected from the passing tests is used to generate models that capture the "correct" behavior of the application. To do so, RADAR leverages (1) Daikon [13], to generate expressions that model values that can be assigned to program variables, and (2) KBehavior [23], to generate finite state automata (FSA) that represent the sequences of statements that can be executed without observing any failure. RADAR checks the traces recorded for the failing executions agains the models derived from the passing executions to identify behavioral differences between original and modified programs that may indicate anomalies and that may be the causes of an observed failure.

MIMIC leverages RADAR's ability to monitor programs and generate and check models for program variables, whereas it does not use RADAR's FSA models. The monitoring, model generation, and model checking capabilities of RADAR have been adapted to fit MIMIC's purpose, as we describe in Section 3.

3. THE MIMIC APPROACH

MIMIC is a failure analysis technique whose goal is to automatically identify the likely causes of failures from a set of failure data (*i.e.,* runtime information about a failure). MIMIC starts from a failure (*e.g.,* a failing test) and produces a list of events that likely explain the cause(s) of the failure. As illustrated in Figure 1, MIMIC works in four steps: execution synthesis, monitoring points detection, anomaly detection, and filtering. We first provide an overview of these steps and then discuss them in detail.

The *execution synthesis* step starts from the failure data and generates multiple executions that mimic the corresponding failure. As we quickly discussed in Section 2.1, in this step MIMIC leverages F^3 [16], which is based on symbolic execution, to synthesize both failing and passing executions. When the approach is successful,

```
602  #define TAB_WIDTH(c_, h_) ((c_) - ((h_) % (c_)))
...
1349 clump_buff = xmalloc (MAX (8, chars_per_input_tab));
...
2680 char *s = clump_buff;
...
2690 if (c == input_tab_char)
       chars_per_c = chars_per_input_tab;
2691 if (c == input_tab_char || c == '\t')
2692 {
2693   width = TAB_WIDTH (chars_per_c, input_position);
2694   if (untabify_input)
2695   {
2696     for (i = width; i; --i)
2697       *s++ = ' ';
2698     chars = width;
2699 }
```

Figure 2: Code example: bug13272 in `pr`.

Passing input: `|||\b|||\t\v\v`

Failing input: `\b\d\b\b\b\b\t\v\v`

Figure 3: Examples of passing and failing inputs generated for program `pr`.

the failing executions mimic the behavior of the original failure—intuitively, they violate the same assertion. The passing executions, conversely, are executions that are similar to the falling ones, in the sense that they follow similar paths and typically reach the point of failure, but do not result in a failure. In its subsequent steps, MIMIC uses these sets of similar executions to determine the events that can discriminate passing from failing executions.

The *monitoring points detection* step determines, using a combination of several heuristics, a set of code locations that are particularly suitable for detecting behavioral differences between passing and failing executions. Because collecting precise behavioral data from all code locations of a program is prohibitively expensive for any non-trivial system, MIMIC selects a small but meaningful set of code locations to be used as monitoring points.

The *anomaly detection* step collects data from the passing executions at the monitoring points determined in the previous step. MIMIC uses the collected data to generate behavioral models that capture the (supposedly) correct behavior of the application. The same monitoring points are then used to collect data from the failing executions. MIMIC uses the data collected from the failing executions and the previously generated behavioral models to identify violations of the models that can indicate an anomaly (*i.e.,* a likely failure cause). As we discussed in Section 2.2, MIMIC leverages RADAR to monitor programs, generate models, and check models.

In principle, the anomalies detected in the previous step could be already reported as likely causes of the failure being investigated. However, some anomalous events could result from spurious behaviors unrelated to the failure. MIMIC addresses these cases in its *filtering* step, which removes from consideration every anomalous event that has not been observed in every failing execution. The rationale for this step is that since all the failing executions fail for the same reason, a valid explanation for the failure should be present in every failing execution that has been analyzed. At the end of this step, the filtered anomalies are the likely failure causes that are reported to the developer to help him debug the application.

Running Example.

To demonstrate how MIMIC works, we use a running example that is also one of the case studies considered in our empirical evaluation. The example consists of a fault affecting `pr`, a Unix utility program for paginating text files before printing. `pr` is part of Coreutils [1]. The considered fault is difficult to trigger, and in fact, it affected many versions of `pr` before being revealed and fixed.

Table 1: Excerpt of potential failure causes produced by MIMIC for `pr`.

Source Line	Model	Actual values
pr.c:2697	input_position ≥ 0	input_position: -5
pr.c:2697	$0 \leq$ input_position ≤ 63	input_position: -5
pr.c:2697	i ≤ 8	i: 13
pr.c:2697	c > i	c: 9 i: 13
pr.c:2697	c $\geq$ i	c: 9 i: 13
pr.c:2697	chars_per_c $\geq$ i	chars_per_c: 8 i: 13
pr.c:2697	chars_per_input_tab $\geq$ i	chars_per_input_tab: 8 i: 13

Figure 2 shows an excerpt of `pr`. The code in the example fails when, before a tab character, the number of backspaces in the input string exceeds the number of characters that can be removed. Figure 3 provides an illustrative example, in which the input string contains a tab (`\t`) at position seven, and six out of the seven characters before the tab are backspaces (`\b`).

The actual failure in the program is triggered when variable `input_position` is assigned a negative value. This variable represents the position in which the next character must be written. When too many backspaces occur in the input string, the program assigns a negative value to `input_position`, intuitively representing the case of a "cursor" that has been moved backward up to a negative position. Processing a tab character causes the program to enter the `if` branch at line 2691. When the value of `width` is computed at line 2693, if `input_position` is negative, macro `TAB_WIDTH` returns a value that is larger than the size of array `clump_buff`. (This macro is apparently designed to work on non-negative values only.) Since `width` exceeds the size of the buffer, the program causes a buffer overflow while iteratively executing line 2697 to map the tab character into a (over-length) sequence of spaces.

MIMIC is able to capture the causes of this failure. For instance, Table 1 shows an excerpt of the failure causes automatically determined by our technique. Column *Source Line* indicates the line of code used as observation point. Column *Model* indicates the property that is systematically violated by all the failing executions. Column *Actual values* shows a sample value extracted from one of the failing executions.

MIMIC suitably identifies the trigger of the failure: negative values assigned to `input_position` (see the first two rows in Table 1). This result might be already sufficient to debug the program, as it is enough to notice that the unexpected negative value assigned to `input_position` causes the assignment of a too large value to `width`, which in turn produces the buffer overflow in the `for` loop at lines 2696–2697. However MIMIC provides additional assistance to developers because it also identifies the unusually high values assigned to variable `i`, which counts the number of iterations that are executed in the `for` loop. This anomalous value explicitly points at the location where the buffer overflow happens, demonstrating that a consequence of the negative value assigned to `input_position` is an excessive number of loop iterations.

In the following sections we discuss how each individual step of the technique works in detail, using our running example for illustrative purposes when possible.

3.1 Execution Synthesis

In the Execution Synthesis step, MIMIC generates a set of passing and failing executions similar to the falling execution that is being debugged. To synthesizes these executions, as we discussed earlier, MIMIC leverages the execution generator in F^3 [16], which can generate executions starting from the sequence of calls performed in the failing execution. It is worth noting that, although we are presenting MIMIC as a technique mainly meant to be applied during in-house debugging, it could also be applied to crash reports from the field, as long as they contain the right kind of failure data.

As we discussed in Section 2.1, the execution synthesis in MIMIC is based on a guided symbolic execution that can generate both failing and passing executions that mimic the failure at hand. Specifically, the failing executions would perform the same calls and fail in the same way as the original failure; the passing executions, conversely, would perform as many as possible of the calls in the call sequence and would not fail. Performing all calls may not necessarily result in a failure, so in some cases F^3 has to skip calls to increase the set of program behaviors it can explore and be able to actually generate passing executions.

To properly distinguish between failing and passing executions, MIMIC relies on an external oracle. When used for in-house debugging, this is not an issue, as it can simply reuse the oracle associated with the failing test case that is being debugged. Even if we wanted to use MIMIC on crash reports from the field, however, it would be possible to define a pseudo-oracle that "recognizes" the reported field failure and can check whether a synthesized execution fails in the same way. The facts that (1) the pseudo-oracle does not have to distinguish failing and passing executions in general, but must simply be able to recognize the observed failure, and that (2) most reports from the field are about program crashes, makes this feasible in many practical cases [15].

Figure 3 shows examples of a passing execution and a failing execution synthesized by MIMIC for the `pr` program. In the figure, $\backslash b$ is a backspace char, $\backslash d$ is a delete char, $\backslash t$ is a tabulation character, and $\backslash v$ is a vertical tab character. The two inputs differ in the number of backspace and printable characters that occur before the tab character, which are exactly the input elements that make the program fail.

The synthesis of many passing and failing executions that represent similar program behaviors is of crucial importance for MIMIC because its capability to detect failure causes depends on the differences that exist between the analyzed passing and failing executions. Usually, the smaller these differences, the more accurate is the detection of the failure causes. Recall that the synthesized failing and passing executions share the majority of the function calls. In our empirical evaluation (see Section 4.4.4) we provide evidence that MIMIC produces definitely better results when working with the synthesized executions than when using the test suites available with the programs.

3.2 Monitoring Points Detection

MIMIC identifies failure causes by comparing the behavior of the program in the failing executions to the models that represent the behavior of the program in the passing executions.

Since collecting data and generating models is an expensive operation, MIMIC automatically identifies a small but effective set of program points that can be used to observe the behavior of the system during failing and passing executions and determine the causes of the failures. To select such proper monitoring points, we defined a *suitability* formula that associates a value in the range $[0, 1]$ to each line in the program. The higher the suitability value, the most likely the program point is a useful observation point.

Given a source line l, the suitability of l is calculated as follows:

$$\textit{Suitability}(l) = \begin{cases} k_{PR} \times \textit{PassRatio}(l) + & \text{l executed by all} \\ k_S \times \textit{Suspiciousness}(l) + & \text{failing executions,} \\ k_{VF} \times \textit{VicinityFailing}(l) + & \\ k_{VPOF} \times \textit{VicinityPOF}(l) & \\ & \\ 0 & \text{otherwise} \end{cases}$$

Since the formula is applied to the generated failing executions, which are all caused by the same fault, the anomalies that are reported to the developer must occur in every failing execution. Therefore, the *Suitability* formula always assigns 0 to the lines of code that are not covered by every failing execution.

If a line l is executed by every failing execution, its suitability is determined as a weighted sum of four terms. The four constants k_{PR}, k_S, k_{VF}, and k_{VPOF} represent the four weights which have to sum to 1. In the following paragraphs we explain the meaning and rationale for each term.

PassRatio(l): The detection of the illegal values assigned to program variables during failing executions requires good models that suitably generalize the behavior sampled with passing executions. If a line is executed by several passing executions MIMIC can generate good models that capture the correct behavior of the application. While if the line is executed by few passing executions, MIMIC might generate models that overfit the executions, failing to fully capture the legal behavior of the application. *PassRatio* represents the ratio of passing executions that cover l, that is the more passing executions cover l the more likely l is selected as a monitoring point. For example, if 5 out of 10 passing executions cover l the *PassRatio*(l) is 0.5.

Suspiciousness(l): To suitably observe the trigger and the effect of a fault, it is important to select monitoring points that are likely to be related to the fault. To this end, MIMIC computes the suspiciousness score of a line of code using fault localization. The higher the suspiciousness score is, the more likely the line of code is faulty or closely related to the fault. In particular MIMIC leverages the customized fault localization technique in F^3, which has been shown to work particularly well for synthesized executions [16], to compute the suspiciousness score.

VicinityFailing(l): The code locations executed by failing executions only are likely to process illegal variable values that result from the failure. To timely capture these anomalous values it is important to have monitoring points that are close to the lines executed by failing executions only. (MIMIC does not directly monitor the lines executed by failing executions only because no passing execution reaches these lines and thus no model of the correct behavior of the application could be generated for these locations.) *VicinityFailing* measures how close a line of code is to a statement executed by failures only. In particular, if L_{only} is the set of lines executed by failing executions only, *VicinityFailing*(l) is defined as

$$\textit{VicinityFailing}(l) = \max_{l_{only} \in L_{only}} \textit{VicinitySourceFile}(l, l_{only})$$

where

$$\textit{VicinitySourceFile}(l_1, l_2) = \begin{cases} 0.7^{\frac{|l_1 - l_2|}{50}} & l_1 \text{ and } l_2 \\ & \text{in same file,} \\ & \\ 0 & \text{otherwise} \end{cases}$$

The vicinity between two lines of code is 0 if the two lines of code are in two different files. Otherwise, the distance is computed using the difference between line numbers. The vicinity is 1 when the two line numbers are the same. Although this simple metric has worked well in the cases we studied, more sophisticated metrics might be needed in the future. One possible solution, if such need were to arise when performing additional experimentation, would be to compute vicinity based on proximity in the control flow graph of the program.

Currently, we compute vicinity using a formula based on an exponential growth to strongly penalize the lines of code that are far from any statement executed by failing executions only. The constants in the formula are chosen to prevent values from dropping too abruptly: the vicinity drops by 30% every time the distance between the lines of code increases by a factor of 50.

VicintyPOF(l): A line of code which is near the point of failure is likely to capture the illegal values that are responsible for the actual failure. To this end we compute the vicinity of a line of code l to the point of failure, namely l_{POF}, using the following formula

$$\textit{VicinityPOF}(l) = \textit{VicinitySourceFile}(l, l_{POF})$$

We did not systematically investigate different combinations of weights to discover the optimal formula for computing the suitability of a line of code l. However, we empirically observed that the *Suspiciousness* and the *PassRatio* terms are the most important ones, with the former term contributing more than the latter in the identification of good monitoring points. We thus defined the following values for the four weights $k_{PR} = 0.25$, $k_S = 0.55$, $k_{VF} = 0.1$, $k_{VPOF} = 0.1$.

The suitability formula produces a ranking of the statements in the program. The statements at the top of the ranking are those that are likely to represent the best observation points for discovering the causes of the failure according to the set of executions available. From the ranking, MIMIC selects the first N non-consecutive statements for monitoring. MIMIC does not select consecutive statements because they are likely to give similar opportunities in terms of their ability to monitor values and discover failure causes. Therefore, if consecutive statements occur among the first N statements, only the one with the highest suitability is selected (the choice is non-deterministic when the suitability is the same).

In practice, the exact value of N depends on the resources that are available (*e.g.,* time available for the analysis, computational power of the analysis infrastructure). In most of our experiments, we used $N = 10$.

MIMIC uses the monitoring points to extensively observe the behavior of the program. In practice, it records the values of all the variables (including fields and elements of data structures) that are visible from the monitoring points. As we discussed in Section 2.2, MIMIC uses the monitoring infrastructure available in RADAR to collect the data [26].

In our running example, MIMIC automatically selected line 2697 among the top 10 lines, which resulted to be an effective choice for determining the causes of the failure—the failure causes reported in Table 1 have been detected using line 2697 as observation point.

3.3 Anomaly Detection

In the Anomaly Detection step MIMIC runs the passing and failing executions and collects information about program behavior at the monitoring points determined in the previous step.

The data collected from the passing executions are used to distill models that capture the values that can be legally assigned to program variables. To this end, MIMIC leverages Daikon [13], which is an inference engine that generates Boolean expressions for a set of variables whose values have been observed in multiple executions. In our case, for each individual monitoring point, Daikon generates a set of Boolean expressions that hold for the variables that can be accessed from the monitoring point.

For the variable values collected from line 2697 in our running example, MIMIC generates both useful expressions, such as `input_position`$\geq$`0` (which captures an important characteristic of passing executions), and spurious expressions, such as `clump_buff` $\geq$ `s` (which does not capture any relevant information about the failure).

To discover anomalous events that might reveal the causes of failures MIMIC checks the data collected while running the failing executions with the models generated from passing executions. Each anomalous event is composed of three fields: the *source line* where the event has been detected, the *model* that has been violated by the anomalous event, and the *actual values* that violated the model. For instance $\langle 2697, \textit{input\_position} \geq 0, \textit{input\_position} = -5\rangle$ and $\langle 2697, \textit{clump\_buff} \geq s, \textit{clump\_buff} =$ " $i\backslash 001"s =$ "$i\backslash 001"\rangle$ are two anomalous events detected by MIMIC for the `pr` case study.

Regarding the classes of faults that can be addressed, this anomaly detection strategy is general. In fact, the actual possibility to address a fault does not depend on the type of fault, rather depends on the impact of the fault on the program variables. In particular, the faults that can be addressed with MIMIC are all the faults that can cause anomalous variable values that can be detected with Daikon invariants (or other invariant generators).

3.4 Filtering

Although MIMIC generates passing and failing executions that are similar by construction, the differences between them might be related to spurious events rather than to events that caused failures. To filter out such spurious events, MIMIC implements a simple heuristic that has shown to be effective in practice: it considers the set of anomalous events detected for each failing execution and filters out the ones that do not occur in *every* failure.

Since the same model might be violated in many different ways, and the way the model is violated is usually irrelevant for identifying the failure causes, MIMIC filters anomalies considering only the models and ignoring the actual values that violate the models. For instance, the model `input_position` $\geq$ `0` at line 2697 of `pr.c` might be violated in many different ways. For a failing execution, `input_position` may be equal to -1, while for a different failing execution its value may be -5. The fact that the two executions witness different values for `input_position` is irrelevant. The important information is that, for all the failing executions, `input_position` has a negative value. For this reason, an anomaly $A = \langle \textit{loc}, \textit{model}, \textit{values}\rangle$ is common to all the failing executions if every failing execution produces an anomaly $A' = \langle \textit{loc'}, \textit{model'}, \textit{values'}\rangle$, with *loc* = *loc'* and *model* = *model'*.

In our running example MIMIC successfully preserved the anomaly $\langle 2697, \textit{input\_position} \geq 0, \textit{input\_position} = -5\rangle$, which is important to understand the fault, whereas it eliminated the anomaly $\langle 2697, \textit{clump\_buff} \geq s, \textit{clump\_buff} =$ " $i\backslash 001"s =$ "$i\backslash 001"\rangle$, which is irrelevant.

The anomalies that are not filtered out are presented to the developer as set of possible failure causes that might explain the reason of the failure reported in the crash report. Since each anomaly that is presented to the developer occurs as many times as the number of failing executions that have been synthesized, there are multiple values that can be presented to the tester to show the way the models have been violated. MIMIC selects one of these values non-

Table 2: Benchmark programs and faults used in our study.

Name	Version	Size (LOC)	Fault Type	Fault ID
`grep`	2.2	10K	Injected	DG_4
`sed`	1.18	14K	Injected	AG_20
`pr`	6.10	3K	Real	[8]
`mknod`	6.10	0.3K	Real	[6]
`od`	6.7	2K	Real	[7]
`XMail`	1.21	1K	Real	CVE-2005-2943

deterministically from the set of available ones, while keeping the whole set of values that violate a model available for inspection. Table 1 shows some of the failure causes automatically reported by MIMIC for our running example.

4. EMPIRICAL EVALUATION

To assess the effectiveness of MIMIC, we implemented a prototype tool and used it to investigate the following research questions:

- **RQ1:** Can MIMIC report anomalies that are related to faults?
- **RQ2:** Does filtering increase the quality of the results?
- **RQ3:** Is the automatic selection of monitoring points effective?
- **RQ4:** Do synthesized executions produce better results than the test suites of the analyzed applications?
- **RQ5:** Can the detected violations help understand the causes of a failure?

In the following we describe our prototype implementation, the objects of study, the experiment setup, and the results of the evaluation.

4.1 Implementation

We implemented MIMIC in a prototype tool written in Java. The MIMIC prototype can analyze failures that affect programs written in C/C++. Our implementation integrates several other tools. In particular, MIMIC uses F^3 to synthesize passing and failing executions from failure data that consist of (partial) call sequences [16] and RADAR to generate and check behavioral models [25]. RADAR monitors applications using the GDB debugger and generates program properties using Daikon [13]. MIMIC collects coverage data using GCOV.

MIMIC uses Daikon with the suppression of redundant models disabled. Otherwise Daikon might incidentally suppress a model that perfectly captures the reasons of the failure when it is implied by other models that do not represent the reason of the failure as clearly as the suppressed one.

4.2 Objects of Study

Table 2 presents the programs that we used to evaluate MIMIC. We selected six real-world open source programs written in C and C++: `grep`, a well known command-line utility for searching the lines that match a given pattern within textual files [20]; `sed`, a stream editor [20]; `Xmail`, a mail server [11]; and three programs from the Coreutils file manipulation package: `pr`, `mknod` and `od` [1]. We selected these programs because they are well-known Unix utilities whose size ranges from 300 LOC to 14 KLOC.

We considered one fault per program. In two cases, `grep` and `sed`, the faults have been defined by a third party (we used the faulty versions of `grep` and `sed` available in the public repository SIR [20]). In all other cases the faults are real faults discovered by end-users in the field. In Table 2, column *Fault Type* indicates the type of fault, while column *Fault ID* provides the exact reference to the fault that has been analyzed.

Each fault is associated with a test case that triggers it and produces the corresponding failure. For real faults, we considered the test case implemented from the report provided by the end-users. For the injected faults, we considered a failing test case randomly selected from the test suites provided with the programs. The failure data have been obtained from running these tests against instrumented versions of the benchmark programs.

4.3 Experiment Setup

In this section we present the design of the empirical studies aimed to answer our research questions.

RQ1: Can MIMIC report anomalies that are related to faults? This research question investigates whether MIMIC can detect the causes of failures, that is, if it can detect anomalies related to the fault under analysis. To answer this research question, we executed the MIMIC prototype on each case study and manually classified the reported anomalies as either true or false positives. We classified as true positives only those anomalies that either represent the condition that triggers the failure or capture erroneous variable values produced by the faulty code. An effective analysis should return few anomalies with a high density of true positives.

RQ2: Does filtering increase the quality of the results? This research question investigates the effectiveness of the filtering step. To answer this research question, we manually inspected the anomalies that MIMIC automatically filtered out and classified them as either true or false positives. An effective filtering should eliminate most of the false positives without eliminating the true positives.

RQ3: Is the automatic selection of monitoring points effective? MIMIC heuristically defines monitoring points taking into consideration the structure of the program and the set of executions that have been synthesized. In principle, the selected monitoring points should guarantee the best observation capabilities for the fault under analysis. However, their effectiveness must be empirically assessed.

To answer this research question, we measured the number of true and false positives obtained by using the observation points determined by MIMIC and by following the common practice of observing the behavior of a program at the entry and exit points of its functions [9,10,23,31]. A good set of observation points should generate more true positives and less false positives.

RQ4: Do synthesized executions produce better results than the test suites of the analyzed applications? This research question investigates the benefit of using a set of synthesized executions that accurately samples the behavior of the application around the failure compared to using the test suites available with the programs. To answer this research question we compared MIMIC as defined in the paper to two alternative configurations.

The first configuration, MIMIC-orig, applies MIMIC using the original test suite instead of synthesized executions, but uses the same observation points computed for the synthesized executions. In this way the differences on the results could be uniquely due to the different test cases used for the analysis. However, using the observation points computed from the synthesized executions for both configurations might penalize the analysis based on the original test suite. We thus consider a third option, MIMIC-orig-obs, which uses the original test suite of the program and the observation points computed for the original test suite instead of the synthesized test suite (*i.e.,* we applied the suitability formula to the tests available with the program instead of the synthesized executions). Since

Table 3: Results for RQ1.

Name	Passing Tests	Failing Tests	Monitoring Points	Failure Causes	
				TP	FP
grep	365	30	10	1	0
sed	27	54	10	0	0
sed	27	54	30	0	0
od	8	4	10	1	10
mknod	53	4	10	3	13
XMail	5650	304	10	8	0
pr	1721	35	10	0	0
pr	1721	35	20	8	21

the test suites available with the programs do not include multiple failing test cases, we used the synthesized failing executions for the purpose of the experiment.

RQ5: Can the detected violations help understand the causes of a failure? This research question investigates whether the results obtained with MIMIC are not only related to the fault (*i.e.,* they are true positives), but are also practically useful to identify the faults. The quantitative investigation of this research question would require a large study based on human subjects. So far, we focused on the technical contribution and such a study, which is motivated by the promising empirical results reported in the next section, is part of our future work. Here we try to address this research question qualitatively by shortly presenting the individual cases that have been investigated and providing qualitative arguments that show how the results returned by MIMIC can help identifying failure causes.

Since MIMIC leverages several computationally expensive techniques (*e.g.,* symbolic execution and dynamic invariants detection techniques) and, more importantly, the goal of the empirical evaluation is to assess the effectiveness of the technique, we did not consider any particular time limit for the analysis. All the cases reported in this paper have been analyzed in few hours, which is compatible with a common "test overnight" scenario. The only exception is the fault in `pr`, which needed one day of processing time due to the complexity of the symbolic states involved in the analysis.

4.4 Results and Discussion

4.4.1 RQ1: Can MIMIC report anomalies that are related to faults?

Table 3 shows the results for RQ1. Columns *Passing Tests* and *Failing Tests* report the number of passing and failing executions automatically synthesized by MIMIC. Column *Monitoring Points* reports the number of program points monitored for each case study. We generally used the top 10 locations. In case 10 was not enough to get any result (see `sed` and `pr`), we tried with a higher number of locations. Columns *TP* and *FP* report the number of true and false positives among the set of the likely failure causes identified by MIMIC.

When using 10 monitoring points, MIMIC detected the failure causes (*i.e.,* $TP > 0$) for 4 out of 6 cases. In two cases, `sed` and `pr`, MIMIC detected no failure causes. To investigate if the problem was related to the number of monitoring points, we repeated the analysis with an increased number of points: 20 monitoring points were enough to identify the failure causes for `pr`, while even 30 monitoring points were insufficient for `sed`. We carefully inspected the fault in `sed` and found that the corresponding fault could not be addressed with MIMIC. This is due to the fact that the faulty code region always fails when executed. There is thus no way to generate passing executions that cover code regions close enough to produce models useful to debug the fault. It is worth noting that, in the cases where the failure cause was not detected, MIMIC generated no false positives either, thus preventing the developer from inspecting useless anomalies.

In some cases, MIMIC generated a perfect outcome, such as for `grep` and `XMail`, where it reported only true positives. For three cases, `od`, `mknod`, and `pr`, conversely, the results included both true and false positives. However, MIMIC always returned a small number of (quick to inspect) anomalies for the cases considered, so we do not expect false positives to be a major issue in these cases. In summary, our results show that MIMIC has the potential to report the causes of failures automatically.

4.4.2 RQ2: Does filtering increase the quality of the results?

Table 4 shows the results for RQ2. For each case, identified by the program name (column *Name*) and the number of monitoring points used (column *Monitoring Points*), Table 4 reports the number of unique anomalies discovered by MIMIC; that is, multiple anomalies that violate the same model in different executions are counted as one. Column *False Positives* shows the total number of anomalies unrelated to the fault detected by MIMIC before applying the filtering step (column *All*), the number of false positives successfully filtered out (column *Filtered*), and the number of false positives that have not been filtered out automatically (column *Reported*). Column *True Positives* shows the total number of anomalies related to the faults detected by MIMIC before applying the filtering step (column *All*), the number of true positives erroneously filtered out (column *Filtered*), and the number of true positives successfully reported to developers (column *Reported*). The last row of the table reports average values across all cases.

The results show that the filtering step can effectively remove many false positives. In fact, it eliminated around 60% of the false positives on average, although with varying performance among the individual case studies. In particular, for 4 out of 7 cases (one case produced no anomalies), filtering eliminated 100% of the false positives, thus returning only useful anomalies to the developers. In the three remaining cases, filtering was less effective, eliminating from 0% to 12.5% of the false positives. This result suggests that, depending on the characteristics of the failure, filtering could have varying effectiveness. In particular, we noticed that filtering was less effective when either (1) MIMIC generated few failing executions (such as for `od` and `mknod`) or it generated many failing executions, but the failure was triggered only through a very specific path with little possibility to obtain diverse failing executions (such as for `pr`). Ideally, MIMIC needs multiple diverse failing executions to effectively filter false positives, and both lack of failing executions and lack of diversity are factors that clearly affect the performance of the filter.

Finally, filtering demonstrated to be conservative, suggesting that the assumptions behind it tend to hold in practice. In fact, filtering never dropped any true positive, as shown in Table 4.

4.4.3 RQ3: Is the automatic selection of monitoring points effective?

Table 5 shows the results for RQ3. Column *MIMIC* shows the number of true positives (column *TP*) and false positives (column *FP*) returned by MIMIC when using the top 10 monitoring points (20 in the case of `pr`) identified by the *Suitability* formula presented in Section 3.2. Column *Entry-Exit Points* shows the number of true positives (column *TP*) and false positives (column *FP*) returned by

Table 4: Results for RQ2.

Name	Monitoring Points	True Positives			False Positives		
		All	Filtered	Reported	All	Filtered	Reported
grep	10	1	0 (0%)	1 (100%)	25	25(100%)	0 (0%)
sed	10	0	-	-	263	263(100%)	0 (0%)
sed	30	0	-	-	315	315(100%)	0 (0%)
od	10	1	0 (0%)	1 (100%)	10	0 (0%)	10(100%)
mknod	10	3	0 (0%)	3 (100%)	13	0 (0%)	13(100%)
XMail	10	8	0 (0%)	8 (100%)	115	115(100%)	0 (0%)
pr	10	0	-	-	0	-	-
pr	20	8	0 (0%)	8 (100%)	24	3(12,5%)	21(87,5%)
Avg			**0%**	**100%**		**59%**	**41%**

Table 5: Results for RQ3.

Name	Monitoring Points	MIMIC Failure Causes		Entry-Exit Points Failure Causes	
		TP	FP	TP	FP
grep	10	1	0	0	0
sed	10	0	0	0	22
od	10	1	10	0	478
mknod	10	3	13	0	0
XMail	10	8	0	8	16
pr	20	8	21	10	10

MIMIC when using the function entry and exit points to observe the program behavior.

The results show that the monitoring points identified with the suitability formula can be used to detect failure causes more effectively than a standard set of monitoring points, such as the entry and exit points of program functions. In fact, when using entry-exit points, MIMIC missed several code locations important for detecting failure causes and the number of successful cases (*i.e.,* cases with at least a true positive) dropped to two.

Monitoring entry and exit points might imply monitoring code locations that are both not well covered by passing executions and unrelated to the problem under investigation, causing the generation of poor models that introduce many false positives. In fact, for three cases out of six (sed, od, and XMail) the strategy based on entry-exit points returned significantly more false positives than MIMIC.

Only in two cases the use of entry-exit points resulted in less false positives. In one of these two cases (mknod), detecting less false positives also implied detecting no true positives; MIMIC, conversely, successfully discovered the causes of the fault in this case. In the remaining case, the entry-exit points strategy produced slightly better results than the strategy implemented in MIMIC, but both approaches were successful.

Overall, this empirical results demonstrate the effectiveness of the strategy defined in this paper, which detects the monitoring points taking into consideration the specific characteristics of the case under analysis, including the failure, the set of synthesized executions, and the structure of the program. In particular, the strategy based on the *Suitability* formula has been able to identify an effective set of monitoring points for all the faults that could be analyzed with MIMIC (recall that the fault in sed cannot be addressed with MIMIC).

4.4.4 *RQ4: Do synthesized executions produce better results than the test suites of the analyzed applications?*

MIMIC discovers failure causes exploiting its unique capability of synthesizing passing and failing executions that are similar to the failing execution under analysis. One key question is to what extent this capability impacts the results. To answer this research question, we compared the results obtained with MIMIC to two alternative configurations: (1) MIMIC-orig, which consists of applying MIMIC to the test suite of the application using the monitoring points computed from the synthesized executions, and (2) MIMIC-orig-obs, which consists of applying MIMIC to the test suite of the application using the monitoring points computed from these same tests. We compare the results in terms of false and true positives.

Figure 4 shows a histogram that illustrates the number of false positives generated by the three configurations for five cases. We did not consider XMail, in this case, because it is not distributed with a test suite (and thus only MIMIC can be applied to it). To keep the histogram readable, we cut the bars at 70. For sed, od, and pr, however, the configurations using the test suite distributed with the applications generated thousands of anomalies. Based on these results, we can conclude that the original test suite cannot be used to analyze these three cases due to the excessive number of reported false positives. On the contrary, MIMIC generated a limited number of false positives for od and mknod, and no false positives for sed, showing the higher effectiveness of the synthesized tests, compared to the original test suites, in the context of our failure analysis.

In the case of grep, the test suite of the program generated again a higher number of false positives compared to MIMIC, which generated no false positives. However, in this case, the number of false positives generated by MIMIC-orig and MIMIC-orig-obs is manageable. The failure in mknod is the only case in which, using the original test suite, the number of false positives that are generated is smaller than (but comparable to) the number of false positives generated by MIMIC.

The results about false positives already show the clear advantage of synthesized executions over the original test suites. The superiority of synthesized executions is confirmed by the data about true positives shown in Figure 5.

In terms of true positives, MIMIC-orig-obs produced the worst results and detected useful anomalies only for grep. When considering MIMIC-orig, the results slightly improved, with the number of cases in which it reported at least a true positive raising to three. In practice, however, only two of these three cases can be practically addressed with MIMIC-orig, as it returned thousands of false

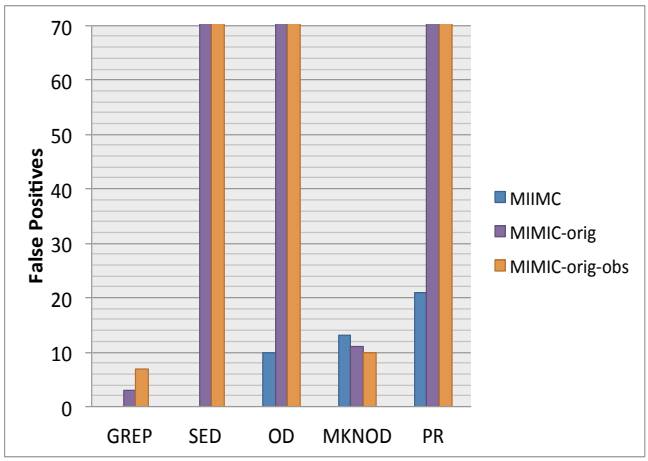

Figure 4: False positives reported by 3 different versions of MIMIC to answer RQ4.

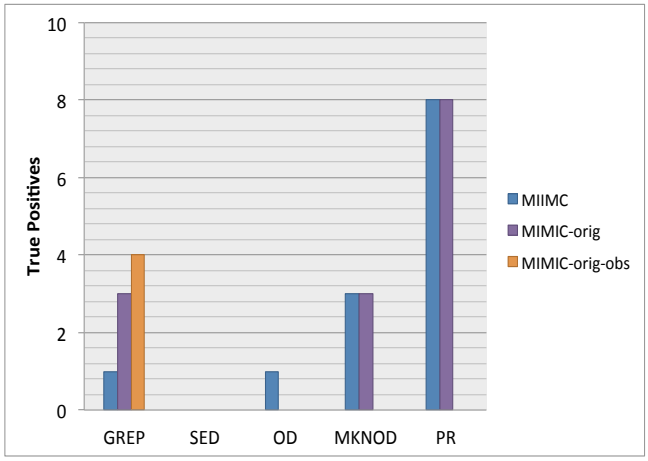

Figure 5: True positives reported by 3 different versions of MIMIC to answer RQ4.

positives for `pr`, which makes the result virtually useless for developers. MIMIC, conversely, returned effective and concise results for four of these five cases.

In summary, when using the original test suite instead of synthesized executions, the effectiveness of the technique decreases significantly. Finally, we were not able to apply MIMIC-orig and MIMIC-orig-obs to `XMail` because `XMail` is not distributed with a test suite (as discussed above), but MIMIC was applicable and effective also in this case.

4.4.5 *RQ5: Can the detected violations help understand the causes of a failure?*

To demonstrate that MIMIC can be practically useful to help developers understand failure causes and debug program failures, we shortly describe the output returned by MIMIC for each analyzed failure, discussing how this output can be used to understand the fault in the program. We will not discuss `pr` because we already presented it as the running example in Section 3. Because of space limitations, we will keep the discussion short and focused on a subset of the anomalies detected by MIMIC.

In `grep`, MIMIC discovered that all the failing executions violate the property $end \geq p$. In the violated model, p is a pointer used to iterate through the locations of a buffer; *end* is also a pointer, but it is used to point at the last location available in the same buffer. This violation indicates that the failure is a buffer overflow caused by a wrong value assigned to p, thus effectively focusing the attention of the developers on the statements that update the value of p. The fault in the program actually consists of a wrong assignment to p.

In `od`, MIMIC discovers that failures happen when the model *file_stats.st_size != n_skip* is violated, that is, when *file_stats.st_size* is equal to *n_skip*. This violation is detected just before the predicate *((uintmax_t) file_stats.st_size <= n_skip)* is evaluated in a conditional statement. Developers are thus directly pointed at the implementation of the two branches following the conditional statement. Exploiting this contextual information, it is fairly easy to recognize that, when *file_stats.st_size* is equal to *n_skip*, the wrong branch is taken by the program—a problem that can be fixed by replacing $\leq$ with $<$ in the conditional statement.

In `mknod`, MIMIC discovers that failures happen when *arg != null* is violated, that is, when *arg* is equal to null. This anomaly exactly captures the failure cause because the program crashes when this null value is passed as a parameter to function *quote*, which is invoked just after the statement that produces the violation.

In `XMail`, MIMIC discovers that the failure happens when predicate $iAddrLength \leq 9$ is violated. This happens, for instance, when *iAddrLength* (a variable whose value is determined by user input) is assigned the value 356. Because variable *iAddrLength* is used to copy chars into a buffer of fixed size, this anomaly indicates a possible stack overflow that happens when *iAddrLength* is assigned a value that is too large, which is exactly the cause of the observed failure.

4.5 Threats to Validity

Even if our results are positive for the cases considered, they might not generalize beyond the systems and failures analyzed in this paper. To mitigate this issue, we considered failures in different applications (both standalone applications and Coreutils) and of different nature (both injected and real faults). However, additional experiments are necessary to assess the generality of the results.

Part of the evaluation exploits the classification of the anomalies as true and false positives. This classification has been the result of a partially subjective work, and there is a risk that different developers might classify the same anomalies in different ways. To mitigate this issue, we classified as true positives only the anomalies that either represent the condition that triggers the failure or capture erroneous variable values produced by the faulty code; we classified as false positives all remaining cases, including the ambiguous ones. To further address this issue, we reported qualitative results that show case by case how the output generated by MIMIC has been useful to locate and understand the faults considered.

A last threat to validity is the possible presence of faults in our tools. To address this threat, we carefully inspected all the anomalies reported by MIMIC for each case considered in the empirical evaluation. By doing so, we gained confidence that our implementation and results were correct.

5. RELATED WORK

In this section, we discuss existing techniques that are closely related to MIMIC. In particular, we discuss statistical fault localization, anomaly detection, experimental debugging, and techniques exploiting synthesized executions.

Statistical fault localization techniques localize faults based on the intuition that code elements executed more frequently by failing executions are more likely to be faulty (*e.g.,* [5, 17, 19, 21, 27]). These techniques differ mainly in the statistical analysis used to compute such likelihood. Most statistical fault localization techniques suffer from two main limitations. The first limitation is that they rely on the existence of a large number of passing and failing test cases [3, 32], which are rarely available in practice. The second limitation is that these techniques provide suspicious code locations without any further explanation, which has been shown to be of limited helpfulness to developers [24]. MIMIC addresses both of these limitations. First, it does not rely on an existing test suite, as it leverages F^3 to automatically generate multiple passing and failing executions from a single failure. Second, it leverages RADAR to infer, from the so generated passing and failing executions, models that capture correct program behavior and violations of such models that can be used to understand the context of a failure and investigate its causes.

Anomaly detection techniques can be used to identify the behavioral anomalies that occur in failing executions [4,12,22,23,26,30]. These techniques infer models that capture the valid behavior of a program from a set of passing executions and identify the anomalous events responsible for the failure by checking the failing executions using the inferred models. The effectiveness of anomaly detection techniques depends on two main aspects: the quality of the passing executions and the strategy used to monitor the program. Most state-of-the-art techniques rely on existing test cases and monitor programs at predefined code locations (*e.g.,* method entry and exit points). A notable exception is RADAR, which can determine the monitoring points dynamically according to the characteristics of the change that is being analyzed. Unfortunately, RADAR can be applied to regression faults only, in the sense that it relies on the existence of an extensive regression test suite [26]. MIMIC is the first anomaly detection technique that (1) generates synthetic executions, instead of relying on existing tests, to explore the execution space close to the original failing execution and (2) selects monitoring points based on the characteristics of both the program and the synthesized executions, rather than a-priori. Results from our empirical studies show that these two capabilities are of fundamental importance in determining the effectiveness of our approach.

Experimental debugging approaches such as delta debugging and predicate switching can also be used to perform fault localization. Delta debugging (*e.g.,* [33]) is based on a divide-and-conqueror algorithm that, given one passing execution and one failing execution, identifies a minimal set of circumstances (*e.g.,* inputs or program states) that can distinguish the two executions, and can thus be considered causes for the failure. Techniques based on predicate switching, conversely, aim to identify the predicates in the code that, if flipped, can transform a failing execution into a passing one [34]. Both delta debugging and predicate switching alter executions in a possibly unsound way, which often results in infeasibility issues that negatively affect the diagnosis ability of these techniques. MIMIC, conversely, always identifies anomalies that differentiate real (*i.e.,* feasible) failing executions from real passing executions.

Other techniques share with MIMIC the idea of applying anomaly detection to automatically *synthesized executions* (*e.g.,* [28, 29]) and differ in both the kind of output they produce and the execution synthesis strategy they use. Sahoo and colleagues use dynamic data dependence information to locate the statements that may have generated anomalous values in failing executions, that is, values that violate invariants derived from synthesized passing executions [29]. The primary purpose of this technique is different from MIMIC's, as it focuses on fault localization and does not aim to detect the causes of failures as MIMIC does. Most importantly, this technique relies on the availability of a grammar-based specification for the generation of test inputs, while MIMIC generates test cases using guided symbolic execution. (These two alternative techniques have somehow complementary strengths and weaknesses, as shown in related work by Kifetew and colleagues [18].) BugEx shares with MIMIC the goal of helping software developers understand the failure context [28]. Specifically, BugEx produces as output a set of predicates that are most likely to occur in failing executions. BugEx and MIMIC differ fundamentally in the way they generate program inputs and the way they observe program executions. BugEx generates inputs from a failing unit test using a search based approach [14], whereas MIMIC uses guided symbolic execution to systematically generate synthetic executions from a set of executions data [16]. As for observing program executions, BugEx monitors a set of state predicates at predetermined points in the program, whereas MIMIC monitors predicates at code locations determined according to the characteristic of both the program and the executions of synthesized tests. Our empirical evaluation shows that this latter approach can be more effective than one based on observing program executions at fixed locations.

6. CONCLUSION

Most automated techniques that aim to help developers localize and understand the causes of software failures suffer from two main limitations. First, they can be applied only when a high number of passing and failing tests are available, which rarely happens in practice. Second, they provide little to no information about the possible causes of a failure, which makes them less useful to developers [24]. To address these limitations, researchers have recently defined techniques that generate passing and failing executions automatically and analyze the differences between passing and failing executions. MIMIC, the approach that we presented in this paper, operates in this space and is the first technique that combines the following unique capabilities: (1) debugging of individual program failures, even when a test suite is not available; (2) generation of passing and failing executions similar to an observed failure and particularly suitable for debugging and differential behavioral analysis; (3) identification of suitable monitoring points for the comparison of passing and failing executions; and (4) effective filtering of false positives by leveraging multiple failing executions.

The results of our empirical evaluation, performed on six faults of several real-world applications, show that MIMIC can effectively detect failure causes. They also show that MIMIC's unique capabilities, and in particular the ability to synthesize passing and failing executions and identify effective monitoring points, are crucial for the success of the technique. These promising empirical results motivate the design of a user study with the goal of confirming that the reports produced by MIMIC can actually help developers locate, understand, and fix faults. In addition to performing this user study, in future work we will also investigate techniques for systematically analyzing each individual anomaly identified by MIMIC to empirically confirm or disprove the correlation between the anomaly and the failure and further improve the effectiveness and accuracy of our approach.

Acknowledgements

This work was partially supported by NSF awards CCF-1320783, CCF-1161821, and CCF-0964647, and by funding from Google, IBM Research and Microsoft Research to Georgia Tech.

7. REFERENCES

[1] Coreutils. http://www.gnu.org/software/coreutils/.

[2] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical Fault Localization for Dynamic Web Applications. In *Proceedings of the International Conference on Software Engineering*, pages 265–274. ACM, 2010.

[4] A. Babenko, L. Mariani, and F. Pastore. AVA: Automated Interpretation of Dynamically Detected Anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 237–248, 2009.

[5] L. Briand, Y. Labiche, and X. Liu. Using Machine Learning to Support Debugging with Tarantula. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 137–146, 2007.

[6] CoreUtils. Fault in Mknod. http://lists.gnu.org/archive/html/bug-coreutils/2008-03/msg00224.html.

[7] CoreUtils. Fault in OD. http://lists.gnu.org/archive/html/bug-coreutils/2007-08/msg00034.html.

[8] CoreUtils. Fault in PR. http://lists.gnu.org/archive/html/bug-coreutils/2008-04/msg00177.html.

[9] C. Csallner and Y. Smaragdakis. Dynamically Discovering Likely Interface Invariants. In *Proceedings of the International Conference on Software Engineering*, pages 861–864, 2006.

[10] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodologies*, 17(2):245–254, 2008.

[11] Davide Libenzi. XMail. http://www.xmailserver.org/.

[12] M. Dimitrov and H. Zhou. Anomaly-based Bug Prediction, Isolation, and Validation: An Automated Approach for Software Debugging. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, 2009.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[14] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the ACM Symposium and the European Conference on Foundations of Software Engineering*, pages 416–419. ACM, 2011.

[15] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the International Conference on Software Engineering*, pages 474–484, 2012.

[16] W. Jin and A. Orso. F3: fault localization for field failures. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 213–223, 2013.

[17] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477, 2002.

[18] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella. Reproducing Field Failures for Programs with Complex Grammar-Based Input. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 163–172, March 2014.

[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[20] Lincoln University of Nebraska. Software-artifact Infrastructure Repository. http://sir.unl.edu/.

[21] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical Model-based Bug Localization. In *Proceedings of the ACM Symposium and the European Conference on Foundations of Software Engineering*, pages 286–295, 2005.

[22] L. Mariani and F. Pastore. Automated Identification of Failure Causes in System Logs. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 117–126, 2008.

[23] L. Mariani, F. Pastore, and M. Pezze. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Transactions on Software Engineering*, 37(4):486–508, 2011.

[24] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 199–209, July 2011.

[25] F. Pastore, L. Mariani, and A. Goffi. RADAR: A tool for debugging regression problems in C/C++ software. In *Proceedings of the International Conference on Software Engineering*, pages 1335–1338, 2013.

[26] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler. Dynamic analysis of upgrades in C/C++ software. In *International Symposium on Software Reliability Engineering*, pages 91–100, 2012.

[27] M. Renieris and S. Reiss. Fault Localization with Nearest Neighbor Queries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 30–39, 2003.

[28] J. Röβler, G. Fraser, A. Zeller, and A. Orso. Isolating Failure Causes Through Test Case Generation. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 309–319, 2012.

[29] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–152, 2013.

[30] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting Object Usage Anomalies. In *Proceedings of the ACM Symposium and the European Conference on Foundations of Software Engineering*, pages 35–44, 2007.

[31] T. Xie and D. Notkin. Tool-assisted Unit Test Selection based on Operational Violations. In *Proceedings of the International Conference on Automated Software Engineering*, pages 40–48, 2003.

[32] Y. Yu, J. Jones, and M. J. Harrold. An Empirical Study of the Effects of Test-suite Reduction on Fault Localization. In *Proceedings of the International Conference on Software Engineering*, pages 201–210. ACM, 2008.

[33] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[34] X. Zhang, N. Gupta, and R. Gupta. Locating Faults Through Automated Predicate Switching. In *Proceedings of the International Conference on Software Engineering*, pages 272–281, 2006.