F3: Fault Localization for Field Failures

Wei Jin
Georgia Institute of Technology, USA
weijin@gatech.edu

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

ABSTRACT

Reproducing and debugging field failures—failures that occur on user machines after release—are challenging tasks for developers. To help the first task, in previous work we have proposed BugRedux, a technique for reproducing, in-house, failures observed in the field. Although BugRedux can help developers reproduce field failures, it does not provide any specific support for debugging such failures. To address this limitation, in this paper we present F^3, a novel technique that builds on BugRedux and extends it with support for fault localization. Specifically, in F^3 we extend our previous technique in two main ways: first, we modify BugRedux so that it generates multiple failing and passing executions "similar" to the observed field failure; second, we add to BugRedux debugging capabilities by combining it with a customized fault-localization technique. The results of our empirical evaluation, performed on a set of *real-world programs and field failures*, are promising: for all the failures considered, F^3 was able to (1) synthesize passing and failing executions and (2) successfully use the synthesized executions to perform fault localization and, ultimately, help debugging.

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

General Terms: Experimentation, Reliability

Keywords: Debugging, fault localization, field failures

1. INTRODUCTION

Due to the limitations of in-house quality assurance activities, and the increasing complexity of software systems, deployed software is bound to generate *field failures*—failures that occur after deployment, while the software is running on user machines. Field failures are notoriously difficult to reproduce and debug in-house, as also shown by a recent survey [43]. To address these issues, both researchers and practitioners have proposed a number of techniques for helping developers recreate and/or understand field failures. Such techniques range from traditional reporting systems, which collect information when a program crashes and send it back to software producers (*e.g.,* [2,3]), to more sophisticated approaches for capturing and analyzing runtime data from deployed applications (*e.g.,* [12,14,17,21,23,27,32]).

In particular, in previous work we proposed BugRedux [23], a general framework that is able to (1) collect different kinds of crash data and (2) use the collected data to synthesize in-house executions that can reproduce (or better, mimic) failures observed in the field. BugRedux collects *crash data* in the form of a list of goals, that is, points in the program that were reached in the given order when the program failed, with the point of failure being the last entry in the list. Intuitively, these goals represent breadcrumbs that must be followed to cause the failure, which is exactly what BugRedux does using an optimized guided symbolic analysis: it tries to synthesize executions that reach all the goals in the right order and generate a failure analogous to the observed one when the last goal (*i.e.,* the point of failure) is reached. Our empirical evaluation of BugRedux, which we performed on a number of real-world failures, showed that BugRedux can be extremely effective at synthesizing failing executions, even when using just a small amount of crash data [23].

Although BugRedux, and other existing approaches, can be very helpful for recreating, in-house, failures observed in the field, they do not provide any explicit support for understanding the recreated failures and their causes. Developers are therefore left with no alternative but to perform traditional manual debugging. To address this limitation, in this paper we present a new technique called F^3 (Fault localization for Field Failures), which extends BugRedux by adding to it support for automated debugging.

We devised F^3 based on several observations we made in our previous work on failure reproduction, and automated debugging in general. The *first observation* is that the most popular fault-localization techniques (*e.g.,* [5, 24, 27]) rely on the existence of numerous passing and failing executions and cannot be applied to individual executions in isolation. This can be a serious limitation in practice because test suites are often of limited size and, especially, rarely contain multiple tests for the same failure. We defined F^3 so that it suitably addresses this limitation. Specifically, we extended our existing execution-synthesis technique so that (1) it generates not one, but rather a set of executions that mimic a failing field execution e and (2) this set includes both passing and failing executions, such that the failing executions fail for the same reasons as e, and the passing executions should be "similar" to e. (We elaborate on the concept of similarity among executions in Section 3.1.) To do so, we first modified the BugRedux algorithm so that it tries to synthesize as many executions as possible for a given set of crash data (*i.e.,* list of goals). In case this set of executions does not contain any passing one, the algorithm starts eliminating goals from the list and tries again to synthesize passing executions. Eliminating goals increases the degrees of freedom of the synthesis, thus increasing the chances of generating passing executions, at the cost of reducing the similarity between the synthesized executions and e, as we explain in Section 3.1.

The *second observation* we made in previous work is that the output of traditional statistical fault-localization techniques, a list of program entities ranked based on their likelihood of being faulty, may be of limited usefulness to developers [31]. Developers tend to give up when the faulty program entity is not among the first ones in the list, which is often the case—even when the number of program entities to inspect before finding the fault is less than 5% of the program, which is considered a good result in most fault-localization literature, that percentage could correspond to hundreds of program entities in any non trivial program. Moreover, developers typically lack *perfect bug understanding* (*i.e.,* they can rarely identify a faulty program entity by simply looking at it), which means that the number of entities they need to examine is even larger.

To address this second limitation, in F^3 we tailored traditional statistical fault-localization techniques by adding to them several customizations and optimizations well suited to the sets of executions generated by our approach and the failures we target. More precisely, as we discuss in detail in Section 3.2, we selected two well known state-of-the-art fault-localization techniques—Ochiai [4] and Observation-Based Model (OBM) [5]—as baseline techniques and defined three optimizations for these techniques, namely, profiling, filtering, and grouping, based on our experience with field failures and our preliminary investigation.

To validate our approach, we implemented our technique in a prototype tool and used the tool to perform several empirical studies on a set of *real-world programs and real field failures*. In our studies, we assessed (1) whether F^3 is actually able to synthesize multiple passing and failing executions for a given set of crash data, (2) whether these synthesized executions can be used for fault localization, and (3) whether our optimizations actually improve the effectiveness of fault localization and to what extent. The results of our studies are promising in that, for all the cases considered, F^3 was able to synthesize sets of passing and failing executions and use these synthesized executions to perform effective fault localization. Our results also show that our optimized fault-localization techniques can mitigate the limitations of their traditional counterparts, at least when applied to the particular set of executions synthesized by our tool. Overall, our results provide initial, yet clear evidence that F^3 can provide developers with information that can help them identify the causes of field failures.

The main contributions of this paper are:

- A novel approach for helping developers with the difficult task of localizing the causes of field failures.
- An implementation of the approach (freely available for download at `http://www.cc.gatech.edu/~orso/software/f3.html`).
- A set of empirical studies that show that F^3 can be effectively used on real-world failures and faults.

2. BACKGROUND

Before discussing F^3, we provide necessary background information on (1) our previous work on field-failure reproduction and (2) the fault-localization techniques that we leverage and extend.

2.1 BugRedux

BugRedux [23] is a general approach for synthesizing, in-house, executions that mimic an observed field failure. More precisely, BugRedux can synthesize, given a program *P*, a field execution *e* of *P* that results in a failure *f*, and a set of crash data *D* for *e*, an in-house execution *e'* with two main characteristics: (1) *e'* results in a failure *f'* that has the same observable behavior of *f* (*e.g.,* it violates the same assertion that *f* violates at the same program location); and (2) *e'* is an actual execution of *P* (*i.e.,* the approach is sound and generates an actual input that, when provided to *P*, results in *e'*).

In the current instantiation of BugRedux, *D* is an ordered list of program points, or intermediate goals, that can be seen as breadcrumbs to be followed to get to the failure. The last goal in the list is the actual failure point. Given this list of goals, BugRedux performs an optimized, guided forward symbolic analysis to synthesize an execution *e'* that reaches the goals in the right order. It then checks whether *e'* fails in the same way as the observed field execution *e* and, if so, provides it to the developer.

Our implementation of BugRedux can collect four different types of crash data, namely, points of failure, stack traces, call sequences, and complete program traces. In our empirical evaluation of the approach, performed on 16 failures of 14 real-world programs, we found that (partial) call sequences are the most useful kind of crash data: using only a small subset—as few as two entries, in some cases—of the call sequence leading to the field failure, BugRedux was able to synthesize in-house executions that reproduced all of the considered failures. Given these earlier results, in this work we only consider crash data consisting of minimized call sequences. More precisely, as we discuss in detail in Section 3.1, we extend BugRedux so that, given a minimized call sequence, (1) it generates not one, but a set of executions that mimic a field failure, and (2) such set contains both passing and failing executions.

2.2 Statistical Debugging

Given the high cost of manual debugging, researchers have investigated and proposed a countless number of automated debugging techniques that can support developers in their debugging tasks. Fault localization in particular, the task of locating the faulty code entities responsible for a failure in a program, has received a great deal of attention in the last decade (*e.g.,* [5, 9, 16, 24, 27]). In this work, we are interested in statistical approaches to perform fault localization, also called statistical debugging techniques. At a high level, these approaches work by observing the behavior of a (ideally) large number of passing and failing executions, performing statistical inference based on the observed behavior, and using the results of such inference to rank program entities in terms of their likelihood to be related to the failure (*i.e.,* their suspiciousness). The program entities would then be presented to the developers in order of decreasing suspiciousness, and the developers would examine every entity in that order (again, ideally) and assess whether that entity is faulty.

To augment our field-failure-reproduction approach with automated debugging capabilities, we considered various statistical fault-localization approaches. Among the numerous approaches presented in the literature, we chose two representative and well-known fault localization techniques by Abreu and colleagues: Ochiai [4] and OBM [5]. We choose Ochiai because previous research showed, both empirically and analytically, that it is more effective than other traditional techniques that use similar metrics [4]. OBM, conversely, is a good representative of a different family of techniques that focus on models of the program behavior. For completeness, we quickly summarize these two approaches.

2.2.1 Ochiai

Ochiai [4] is a spectra-based fault-localization technique that leverages coverage information in passing and failing runs to compute the suspiciousness of program entities and rank them accordingly. The specific formula used by Ochiai was previously used for computing genetic similarity in molecular biology. Like most spectra-based fault-localization techniques, Ochiai assigns a suspiciousness value to each program entity based on coverage information and on the following intuition: the higher the number of failing runs that executed a program entity, the higher its suspiciousness; conversely, the higher the number of passing runs that executed a program entity (or the higher the number of failing runs that did not execute a program entity), the lower its suspiciousness. Accordingly, the formula used by Ochiai to compute suspiciousness for a program entity *en* is the following (using the same notation used in Reference [4]):

$$suspiciousness(en) = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \times (a_{11} + a_{10})}}$$

In the formula, a_{11} indicates the number of failing executions that exercised *en*, a_{01} indicates the number of failing executions that did not exercise *en*, and a_{10} indicates the number of passing executions that exercised *en*. In theory, the approach can be instantiated for any type of program entity (*e.g.,* statements, functions, branches, or predicates).

2.2.2 OBM (Observation-Based Model)

OBM [5] models the executions in a way that is different from that used in spectra-based fault-localization techniques. Specifically, OBM considers sets of one or more entities in the program as independent diagnoses d_k for a failure *f* (*i.e.,* the entities in a d_k set represent possible causes of *f*). Each execution is then treated as an observation *obs* of the system that can be used to confirm or refuse a set of diagnoses. To do so, OBM uses the following Bayesian formula, which computes the conditional probability of each diagnosis to be a possible cause of the failure:

$$Pr(d_k|obs) = \frac{Pr(obs|d_k)}{Pr(obs)} Pr(d_k).$$

Since OBM assumes that program entities fail independently, the prior probability that a diagnosis is correct, $Pr(d_k)$, and the prior probability that an execution is observed, $Pr(obs)$, are constants and identical for all d_ks. The posterior probability for each observation conditional to a diagnosis, $Pr(obs|d_k)$, is computed based on whether *obs* is a passing or a failing run. The conditional probability of each individual diagnosis d_k is the product of $Pr(d_k|obs)$ over all observations, as observations are considered to be independent. These conditional probabilities represent the likelihood of a diagnosis to contain the causes of the failure.

3. THE F^3 APPROACH

As discussed in the Introduction, the overall goal of this work is to help developers locate the likely causes of an observed field failure using some form of statistical fault localization. Figure 1 provides a high-level overview of the approach we propose, which we call F^3 (Fault localization for Field Failures).

As the figure shows, given (1) a set of minimized crash data produced by BugRedux for a field failure *f* (*i.e.,* a list of goals leading to *f*) and (2) the failing application, F^3 produces a report of *likely faults*: a list of program entities ordered in terms of likelihood of being faulty and being responsible for *f*. The report can then be used by developers to investigate the causes of field failure *f*.

F^3 consists of two main parts: the *Execution generator* and the *Fault localizer*. The execution generator component extends our BugRedux approach so that, given a set of crash data for an execution *e* that fails with failure *f*, it synthesizes two sets of executions: *FAIL*, a set of failing executions that mimic *e* and generate a failure analogous to *f*, and *PASS*, a set of passing executions that are "as similar as possible" to *e* but do not fail (see Section 3.1 for more details on this concept of similarity among executions). The fault localizer component performs our optimized statistical fault localization techniques on these two sets of executions to identify a set of program entities that are likely to be responsible for failure *f*. In the rest of this section, we discuss in detail these two components.

3.1 Execution Generator

One major issue with traditional statistical fault localization is that the ranking results highly depend on the quality of the test cases (or, more generally, executions) available for the statistical analysis. In general, statistical fault-localization approaches require a large number of passing and failing executions to be effective. Moreover, the quality of the executions is also important. If, for instance, the passing executions exercise completely different parts of the code than the failing executions, they are of limited usefulness for the analysis. Unfortunately, as we mentioned in the Introduction, it is rarely the case that such a high-quality set of executions is available in practice, and identifying inputs that can generate suitable executions is a non-trivial task [33]. In the case of field failures, in particular, the state-of-the-art failure reproduction techniques (*e.g.,* [23, 37]) generate at most one failing execution, preventing any kind of statistical analysis.

Our intuition is that this problem can actually be seen as a missed opportunity. Existing execution synthesis approaches can be extended to generate a suitable set of both passing and failing executions to be used in fault localization. Furthermore, because of the way they are generated, these executions may be even more amenable to statistical fault localization than existing test suites.

As explained above, our execution generator takes as input an ordered list of program locations *l*. In this context, *l* is a minimized list of goals that contains enough entries to guide the generator and allow it to synthesize an execution that mimics the observed field execution *e* and reproduces the observed field failure *f* [23]. In our original BugRedux approach, *l* was used to generate a single failing execution. In F^3, we extend our original approach so that it continues to synthesize executions that mimic the one observed in the field until it reaches a given time limit.

By construction, each execution that is successfully synthesized in this way is guaranteed to reach all program locations in *l* and reach them in the right order. Therefore, all these executions reach the point of failure, share with *e* a set of intermediate execution points, but are likely to follow different paths than *e*. It is worth noting that reaching the point of failure does not guarantee that the program would fail at that point, as that also depends on the state of the execution. Therefore, some of the synthesized executions may not result in failure *f*. Although this is true in general, in practice it is often the case that *l* provides enough constraints to the execution synthesis algorithm that all generated executions would indeed trigger the failure, and no passing execution would be generated.

In this case, our technique increasingly eliminates entries from *l* and tries to synthesize executions using this reduced list *l'* as a guide instead of *l*. (In our current instance of the approach, we use a straw-man approach that simply eliminates one entry at a time starting from the beginning of the list.) The rationale for this approach is that eliminating entries in the list augments the degree of freedom of the synthesis algorithm, which in turn increases the chances of generating passing executions. Generating executions using a smaller list, however, is bound to reduce the degree of similarity between the synthesized executions and the original execution *e*. Therefore, the generated executions will be increasingly dissimilar from *e*, and thus less likely to be useful for fault localization, as more entries are eliminated from the list. Fortunately, as the results of our empirical evaluation in Section 4 show, this approach seems to work well in practice: the execution synthesizer did not have to eliminate many entries to generate passing executions, and the degree of similarity between the synthesized executions and the original execution was not considerably affected.

This process terminates when a time limit is reached, one or more passing executions have been successfully synthesized for a

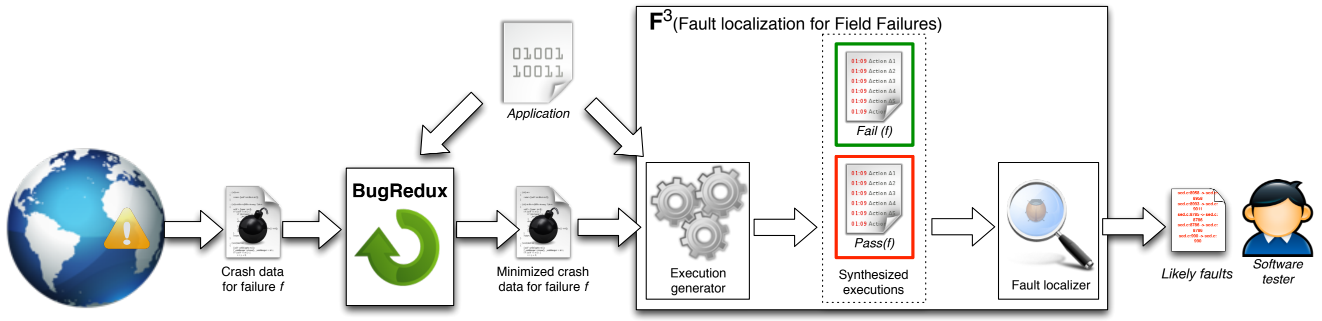

Figure 1: High-level overview of F^3.

given sublist, or no more entries can be eliminated from *l'* (*i.e.,* the list is empty). When successful, the execution generator would therefore produce the two sets of executions *FAIL* and *PASS* mentioned above, which contain failing and passing executions. These executions would be similar, in that they share a set of intermediate execution points, because all of them are derived from *l* or a subset thereof. Intuitively, using such similar passing and failing runs can help fault localization. In fact, recent research has shown that generating passing executions that are close to failing executions can considerably improve the results of fault localization techniques [33, 35].

3.2 Fault Localizer

In this section, we discuss how F^3 uses the sets *FAIL* and *PASS*, synthesized by the execution generator and discussed in the previous section, to perform fault localization and identify the code that may be responsible for the field failure *f*.

There are many ways to perform fault localization given two sets of failing and passing executions, as a variety of statistical fault localization techniques, and variations thereof, can be used to this end. As we discussed in Section 2.2, we chose Ochiai and OBM as representative techniques and suitably customized and optimized them. Specifically, based on our previous experience and our initial results, we customized these techniques along three main directions: use of profiling information, aggressive filtering, and grouping of related entities. In the next sections, we describe our optimizations and discuss how our fault localizer uses the customized fault-localization techniques.

3.2.1 *Traditional Fault Localization*

One straw-man way to use sets *FAIL* and *PASS* for debugging is to simply apply traditional statistical fault localization techniques to these sets. Both fault-localization techniques we are considering only need two such sets to operate and can be used without any additional modification. However, we defined F^3 based on the intuition, later confirmed by the empirical results presented in Section 4, that we can improve the effectiveness of traditional fault localization by suitably tailoring these techniques. To do so, we defined the three customizations of these techniques discussed in the rest of this section.

3.2.2 *Fault Localization with Filtering*

One problem with traditional fault-localization approaches is that most of them rely on a potentially misleading metric for defining and evaluating the approach: they measure effectiveness based not on the number of program entities developers must inspect before identifying the fault, but rather on the percentage of the program they must inspect. Although having to inspect only 5% of the program may appear as a fairly positive result at first glance, considering that this may correspond to hundreds or thousands of program entities gives a quite different, and more practical, perspective. In fact, a human study we performed in previous work provides clear evidence that fault localization techniques should focus on improving *absolute rank* rather than percentage rank [31], as developers are likely to stop inspecting the list of suspicious program entities if they could not identify relevant entities among the first few entries. (Some researchers have proposed to address this issue by cutting the ranked list at a given size or at a given suspiciousness threshold, but that introduces the issue of finding the right size or threshold [18, 33].)

This issue is made even worse by the fact that most approaches tend to assume *perfect bug understanding*, that is, they assume that simply examining a faulty statement in isolation is always enough for a developer to detect, understand, and correct the corresponding bug. Our aforementioned human study [31] also shows that this is not a realistic assumption, which means that the number of entities developers actually examine is normally larger than the number of entities they have to examine in the ranked list.

Because F^3 can generate many passing and failing executions that are similar and tend to focus on a relatively small part of the program, we believe that we can aggressively filter the information computed using these executions in a way that would not be as effective if used on passing and failing executions that are not specifically created to be similar.

Such an aggressive filtering of the list of suspicious program entities has multiple advantages. First and foremost, it can improve the absolute rank of the faulty program entities, which can greatly benefit fault localization. Second, it can also reduce the total length of the ranked list of program entities to examine in a principled way, that is, not based on the choice of an arbitrary size or suspiciousness threshold [18, 33]. Finally, filtering can reduce the number of entities to be considered in the statistical analysis, which can make the analysis more efficient. (This latter advantage would be generally marginal, however, unless a particularly expensive statistical analysis is used.)

F^3 performs filtering using dynamic information about the program entities (branches, in our case) executed in both passing and failing runs, with the goal of discarding beforehand parts of the program that are likely to be irrelevant for the failure. Specifically, we defined three different types of filters, that we use to exclude from the statistical analysis some of the branches in the program. That is, if we define the set of branches exercised by an execution e_i as

$$BR(e_i) = \{br_{i1}, br_{i2}, ..., br_{im}\},$$

using filter FIL_x corresponds to considering only the following branches for the statistical analysis:

$$BranchesToAnalyze = FIL_x \cap \bigcap_{e_i \in FAIL \cup PASS} \{BR(e_i)\}$$

The *first type* of filter we defined considers only branches that are exercised in *all* failing executions and exclude all other branches from the statistical analysis:

$$FIL_f = \bigcap_{e_i \in FAIL} BR(e_i)$$

Clearly, FIL_f may contain branches that are executed by passing executions too, and in some cases by all passing executions, such as initialization code. Therefore, to further filter the fault localization results, F^3 considers a *second type* of filter, FIL_{fp}, which is more aggressive because it further excludes from the analysis those branches that are exercised in *all* passing executions:

$$FIL_{fp} = FIL_f - \bigcap_{e_i \in PASS} BR(e_i)$$

The rationale for FIL_{fp} is that by removing common branches in both passing and failing executions, we remove the branches that may be visited by all executions of the program and are thus likely (but obviously not guaranteed) to be irrelevant.

The *third type* of filtering in F^3 further reduces the amount of entities considered in the statistical analysis by also ignoring branches on which other branches in FIL_{fp} are control dependent:

$$\begin{aligned} FIL_{dep} = FIL_{fp} - \{ & br_k \mid br_k \in FIL_{fp} \wedge \\ & \exists br_j \in FIL_{fp}, j \neq k, br_j \text{ is c.d. on } br_k \} \end{aligned}$$

The rationale, in this case, is that branches that control other branches are often (but clearly not always) responsible for reaching those branches and not directly responsible for the failure.

The three filters we defined are obviously increasingly restrictive (*i.e.,* $FIL_{dep} \subseteq FIL_{fp} \subseteq FIL_f$) and can thus be used to perform an increasingly aggressive filtering. Note that, although there is overlapping between the information used to filter and the information computed by statistical fault localization, filtering still provides an independent way of improving the results of fault localization because it can reduce the overall number of suspicious program entities considered, as we show in Section 4. Moreover, we expect these filters to be particularly effective when operating on similar passing and failing executions, such as the ones generated F^3. First, F^3 can generate several failing executions for a given fault, rather than just one, which we found to be typically the case in existing test suites. This can help narrowing down the possible location of the fault. Second, F^3 can also generate several passing executions that share commonalities with the failing ones, unlike in typical test suites, where passing and failing executions tend to be quite different from one another. This can help filtering out irrelevant entities.

3.2.3 Fault Localization with Profiling

Most fault-localization approaches are based on the concept of coverage of a given program entity, where the entity is typically a statement, a branch, or a more generic predicate. If we consider the formula used to compute the suspiciousness of an entity en in Ochiai, for instance, which we showed in Section 2.2.1, the value a_{11} would be increased by one for each failing execution that exercises en. In our initial experience with F^3, however,we have observed that coverage is not necessarily always the best kind of dynamic information for fault localization. Consider, for instance, the code snippet in Figure 2, which is a simplified version of a real bug we encountered in one of the programs we studied.

In this example, the program reads characters from the input stream until a newline character is encountered. The code contains a buffer-overflow bug, which is triggered when the size of the input stream is greater than 10. Both passing and failing executions could cover line 4, with the difference that failing executions

```
   ...
1. int a[10];
2. int i=0;
3. do {
4.     a[i] = getchar();
5.     i++;
6. } while (a[i-1]!='\n');
   ...
```

Figure 2: A code snippet containing a buffer-overflow bug.

would execute that line more than 10 times. From the standpoint of a technique based on coverage, however, statement 4 appears in every passing and failing run and is therefore not more suspicious than any other statement executed in all runs.

In general, we observed that many failures depend on the number of times one or more program entities are executed. Therefore, in our approach, we also consider an extension of Ochiai that uses profiling, rather than coverage information. (We consider only Ochiai because, due to the way OBM is defined, it is not possible to consider profiling information in OBM without redefining the way in which it computes conditional probabilities for each diagnosis.)

To extend Ochiai with profiling information, we replace the original suspiciousness formula for an entity en with the following one:

$$suspiciousness(en) = \frac{p_{11}}{\sqrt{(p_{11} + p_{01}) \times (p_{11} + p_{10})}}$$

In this new formula, p_{11} indicates the number of times all failing executions exercised en, instead of the number of failing executions that exercised en, as in the original formula. Similarly, p_{10} indicates the number of times all passing executions exercised en, and p_{01} indicates the number of failing executions that did not exercise en. Using this new formula, a program entity en with higher p_{11} and lower p_{10} would be consider more likely to be faulty.

Besides modifying the formula for computing suspiciousness values, in our definition of Ochiai we also introduce an additional optimization that aims at breaking ties for entities with the same suspiciousness value: when two entities have the same suspiciousness value, we order these entities based on the value of p_{11}. This optimization, which could be used also with the original Ochiai approach, accounts for situations in which the effect of p_{11} is masked (*e.g.,* when both p_{01} and p_{10} are zero), which occurred several times in our experiments.

Let us consider again the buggy code snippet in our example of Figure 2. If we apply our customized version of Ochiai to that code, the branch that leads to statements 4 and 5 would be (correctly) ranked higher than other entities, unlike what happens when using coverage information.

3.2.4 Fault Localization with Grouping

Current instantiations of statistical fault-localization techniques usually instrument low-level code to collect the runtime information used for their statistical analyses. As a result, statements that appear in a single source code location (*i.e.,* a single line in the source code) are commonly split into multiple low-level instructions (and even multiple basic blocks). Because developers would inspect a program at the source code level, and thus consider statements from the same source code location together during debugging, low-level entities should be suitably grouped when reporting the fault localization results to developers. To do so, we leverage the source code information in the low-level program entities and use a simple heuristic: if program entities that have the same suspiciousness value correspond either to the same line of source code or to consecutive lines, we group them together and report them to developer as a single entry.

This optimization is different from the previous two optimizations, as grouping mainly helps developer understand and consume the ranked list produced through statistical fault localization. In other words, this optimization addresses an issue that is mainly related to the *engineering*, rather than the *definition*, of the approach. We discuss it here nevertheless because we have observed that it can make a considerable difference in practice and believe it may apply to other related fault-localization techniques. Without grouping, (low-level) related program entities may be spread among other entities with the same suspiciousness value, and the developers would likely waste time going back and forth in the list before they understand the relationships among these entities.

4. EMPIRICAL EVALUATION

To assess the practical usefulness of our technique, we implemented it in a prototype tool and applied the tool to a set of ten real-world programs and corresponding failures. More precisely, we investigated the following research questions:

- **RQ1:** Can F^3 synthesize multiple passing and failing executions for a given set of crash data?
- **RQ2:** Can F^3 use these synthesized executions to perform fault localization effectively?
- **RQ3:** Do our optimizations actually improve the effectiveness of fault localization and, if so, to what extent?

In the rest of this section we discuss the implementation of F^3, our experimental protocol, and the results of our evaluation. For ease of presentation, hereafter, we use the name F^3 to refer to the implementation of our approach.

4.1 Implementation

F^3 works on C programs and is built on top of the BugRedux tool [23], whose functionality it leverages and extends. The first component of F^3, the execution generator, leverages the symbolic execution engine KLEE [11], customized to (1) use crash data as a guide for its search and (2) generate both passing and failing runs. The fault localizer leverages the LLVM compiler infrastructure (`http://llvm.org/`) to add to the code probes that record various coverage and profiling information. LLVM is also used to compute the static control dependence information necessary to calculate set FIL_{dep}, which we discussed in Section 3.2.2. Finally, we implemented the fault localization approaches considered and their optimizations as Perl scripts that analyze the collected execution traces. F^3 performs fault localization at the basic-block (rather than statement) level and identifies a basic block by means of the branch leading to it. We accordingly report fault-localization results in terms of branches, and thus basic-blocks identified by those branches. We use branches for simplicity, as we also use them for filtering. Note that branch information subsumes block—and thus statement—information, and in fact branches are actually directly used in some fault-localization techniques (*e.g.,* [26,27]).

4.2 Objects of Study

For this study, we selected programs that we and other researchers used in previous work (*e.g.,* [8,15,23]). These are *real-world*, non-trivial programs that contain known faults and are available, together with a test suite, from two public repositories: SIR [1] and exploit-db [8]. The three faults in the programs from SIR were seeded by researchers and were designed to be representative of different types of real faults. The seven faults from exploit-db, conversely, are *real faults* reported by users in the field (*i.e.,* exactly the kinds of *real field failures* that our technique targets).

Note that, for convenience, to avoid bias, and similar to what we and others did in previous work (*e.g.,* [23,29,37]), we only selected programs with crashing bugs. Although there are ways to identify

Table 1: Programs from SIR and exploit-db used in our study.

Name	Repository	Description	Size (kLOC)	# Faults
gzip	SIR	compression utility	5	1
grep	SIR	pattern-matching utility	10	1
sed	SIR	stream editor	14	2
aspell	exploit-db	spell checker	0.5	1
xmail	exploit-db	email server	1	1
htget	exploit-db	file grabber	3	1
socat	exploit-db	multipurpose relay	35	1
rsync	exploit-db	file synchronizer	67	1
exim	exploit-db	message transfer agent	241	1

non-crashing failures, such as anomaly-detection techniques or assertions, crashes are commonly targeted failures because they can be effectively treated as built-in, completely objective oracles. In order to focus on interesting faults, we also excluded all programs whose faults could either be easily revealed through exhaustive exploration of the program space (*i.e.,* without any guidance) or easily found because in close proximity to the point of the crash (*i.e.,* a simple analysis of the crash stack would allow for localizing these faults). In summary, we believe that all of the ten faults, and corresponding failures, that we selected are sophisticated enough that (1) state-of-the-art in-house testing techniques would miss them, and (2) developers would have a difficult time investigating them with only limited information from the field.

Table 1 shows the relevant information about the programs and faults we considered. For each program, the table shows its name, repository of provenance, size, and number of faults. For the faults in the exploit-db programs, we identified the location of the faulty entities using documentation and bug fixes available for these programs. For the seeded faults in the SIR programs, we used the positions of the seeded faults as fault locations.

4.3 Experiment Setup

To collect our experiment data, for each fault considered we proceeded as follows. First, to simulate the field failure for a program from SIR, we executed the test cases distributed with the program until we found a test case *t* able to cause a program crash. For each of the programs from exploit-db, conversely, we directly executed the failing test case *t* provided for each failure (uploaded to the repository by the real users who reported the failure). We used *t* as a proxy for the field failure to be investigated, and executed an instrumented version of the program against *t* to collect the crash data CS_{orig}. We then provided CS_{orig} to BugRedux, which generated the corresponding minimized crash data CS_{min}. (Note that this was done for convenience, as an optimized implementation of our approach would integrate the minimization and execution-generation phases.)

Next, we used F^3's execution generator to try to generate the sets of failing and passing executions *FAIL* and *PASS*, using a time threshold of five hours. This step could terminate with one of two possible outcomes: the execution generator was either (1) able to create two non-empty sets *FAIL* and *PASS* or (2) unable to synthesize any passing execution using CS_{min}. In this latter case it would try again after eliminating entries from CS_{min}, as discussed in Section 3.1 (in the worst case, the execution generator would synthesize passing executions using an empty set of crash data, that is, with no guidance that could make the execution similar to those generated using CS_{min}).

We then collected complete program traces for each execution in the *FAIL* and *PASS* sets and computed the ranked list of suspicious entities using both the unmodified fault-localization techniques considered and our optimized versions. As discussed in Section 4.1, our implementation operates at the branch level, so

Table 2: Number of failing and passing executions generated by F^3. For the programs without a star, passing executions are generated by CS_{min}; for the programs with a single star, passing executions are generated using a subset of CS_{min}; and for programs with two stars, passing executions are generated using an empty list, that is, without guidance.

Faults	CS_{min}	# failing	# passing
exim	326	598	4
xmail	363	303	1001
sed.fault2	7	54	30
sed.fault1*	12	1017	296
grep*	2	567	137
aspell*	256	134	10
htget*	2	44	210
gzip.fault2**	2	5	27
socat**	3	46	5
rsync**	2	156	2576

when a branch is filtered out, all statements control dependent on that branch are also eliminated.

Finally, to be able to assess the effectiveness of each individual optimization, we computed our results first with one optimization enabled at a time, and then with all optimizations enabled. We ran all experiments on a machine with a 2.66 GHz Quad Core Intel i7-920 processor and 6GB of memory running Linux Ubuntu 11.04.

4.4 Results and Discussion

This section presents the results of our empirical study and discusses their implications in terms of our three research questions.

4.4.1 Execution Generation

Table 2 shows the execution generation results. For each fault considered, identified by a unique fault ID, the table reports the number of entries in the minimized crash data (*i.e.,* the number of goals in the list fed to the execution generator) and the number of failing and passing executions synthesized by F^3 using this data.

As the table shows, our execution generator was able to synthesize both passing and failing executions for all faults considered. However, we can divide the faults in three groups, based on how these executions were synthesized. For three of the faults, the ones not marked, the generator was able to generate passing and failing executions using CS_{min}. For four other faults, those marked with a single star, the generator had to use a reduced list of goals to generate passing executions because all of the executions synthesized using the original CS_{min} were failing ones. For the remaining three faults, marked with two stars, the generator had to use an empty list of goals to generate passing executions. According to our intuition, and as we discussed in Section 3.1, we expect the degree of similarity between the synthesized passing executions and the originally observed one to decrease as we go from the first group to the third. We discuss how this degree of similarity affects the effectiveness of the fault localizer when we present the fault-localization results.

Overall, F^3's execution generator was able to generate passing and failing executions for all ten programs, and corresponding failures, considered using CS_{min} or a subset thereof. We can therefore answer our first research question in a positive manner.

Answer to RQ1: *For the programs and failures considered,* F^3 *was able to synthesize multiple passing and failing executions.*

Before moving to RQ2, and discussing whether the generated executions can be useful for debugging, it is worth clarifying an important point. Because in several cases the number of entries in CS_{min} is extremely low, it is legitimate to wonder whether the program points in such CS_{min} can be used directly for fault localization. Intuitively, if program entities in CS_{min} were to provide enough guidance to the execution generator to allow it to reproduce the failure at hand, they could provide enough information to locate the fault(s) causing the failure. We investigated whether this was the case by manually checking the entries in all CS_{min} sets with size 20 or less. We found that, although such entries are useful to guide the execution towards program regions that contain the faulty code, they have a subtle and indirect connection with the exact location of the faults. In one case, for instance, the entry was in the same basic block of an assignment whose effect allowed the execution to reach the faulty code in a completely different part of the program.

Table 3: Ranks of the faulty entity (over the total number of entities reported) using F^3's fully optimized fault-localization techniques.

Faults	Ochiai+	OBM+
exim	1/3	1/3
xmail	1/3	1/3
sed.fault2	1/11	8/11
sed.fault1*	13/19	13/19
grep*	12/72	12/72
aspell*	-/0 (1/45)	-/0 (6/45)
htget*	-/0 (1/93)	-/0 (67/93)
gzip.fault2**	3/80	49/80
socat**	11/14	11/14
rsync**	6/28	6/28

4.4.2 Fault Localization

To answer RQ2, we applied F^3's fault localizer to the sets of passing and failing executions synthesized by F^3's execution generator and summarized in Table 2. Table 3[1] presents the results of this study. The columns in the table show the fault ID and the results of applying our customized versions of Ochiai and OBM with all three optimizations enabled, indicated as Ochiai+ and OBM+, to the corresponding failure. The results are presented as the absolute position of the actual fault in the ranked list of suspicious entities produced by the fault localizer over the size of that list.

Such position represents the number of program entities that developers would have to examine before finding the fault. For exim, for instance, F^3 generates a ranked list of three suspicious entities, and developers would have to examine only the first one, which corresponds to the basic block that contains the actual fault. Note that, for cases in which the faulty entity had the same suspiciousness value as other entities, we reported in the table the worst-case result, that is, the case in which the actual fault was placed last in the list among the entities with the same suspiciousness.

In two cases, aspell and htget, the filtering was too aggressive and eliminated all suspicious entities from the list. For these cases, we also report, in parentheses, the position of the fault in the list (over the total size of the list), before filtering was performed. We believe this is justified because, in cases in which the list of entities after filtering is empty, the sensible course of action would be to report the ranking computed without filtering.

Looking at the results in the table, we can make several observations. One first observation is that F^3's fault localizer, when operating on the synthesized passing and failing executions produced by the execution generator, is in most cases quite effective: for 2 out of 10 cases (3 in the case of Ochiai+), the real faulty entity is the first entry in the ranked list; and for another 4 cases (5 in the case of Ochiai+), it is within the first 15 entities in the list. Moreover, even for aspell and htget, the faulty entity is in position 1 when us-

[1]In this and in the following tables, the annotations no star, one star, and two stars have the same meaning they have in Table 2 and are repeated for the reader's convenience.

ing Ochiai+ without filtering. Overall, the only negative results are for htget and gzip.fault2 when using OBM+, for which the faulty entities are ranked in position 67 and 49. In all other cases, F^3's fault localizer produces quite encouraging results.

Another observation we can make looking at the table is that there is some indication that the degree of similarity between the synthesized passing executions and the original failure does seem to affect the effectiveness of fault localization, which confirms our intuition. For the three faults for which F^3 was able to generate passing executions using the complete list of goals (CS_{min} in Table 2), the fault-localization techniques was able to rank the faulty entities in the first position in most cases. For faults in the second and third groups, for which F^3 generated passing executions using a (possibly empty) subset of goals, the fault localization techniques did not perform as well.

To get more insight on this aspect of the technique, we manually checked the ranked lists generated by F^3. Interestingly, we found that, in most cases, the fault localizer did assign the faulty entity the highest suspiciousness value. However, other program entities related to this faulty code were also assigned the same value. (Because we report the worst-case result, as described above, this lowered the rank of the faulty entity that we report.)

One final observation we can make about the results is that, although the performance of the two fault-localization techniques is similar, Ochiai+ seems to be more promising, as it produces consistently better results. However, we will need to perform further experimentation to confirm this result.

Overall, F^3's fault localizer was in most cases effective when applied to the passing and failing executions synthesized by the execution generator, which lets us answer also our second research question in a positive manner.

Answer to RQ2: *For the programs and failures considered,* F^3 *can leverage the execution generator's synthesized executions to perform fault localization effectively in most cases.*

4.4.3 Effectiveness of Our Fault Localization Optimizations

To answer RQ3, in this section we study the results of applying our three fault-localization optimizations separately, so as to assess the improvement made by each individual optimization.

Filtering. Before discussing the extent to which filtering can improve fault localization for field failures, in Table 4 we show the size of the filtering sets (see Section 3.2.2) computed by F^3 for the programs and faults considered in our study. For each fault considered, the table shows the total number of branches exercised by all failing and passing executions (Total) and the size of the FIL_f, FIL_{fp}, and FIL_{dep} sets. Because these sets are increasingly restrictive, their sizes decrease when going from FIL_f to FIL_{dep}. Intuitively, the numbers in the table show how aggressive are the different filters in reducing the amount of code to be considered by statistical fault localization. For xmail, for instance, all failing and passing executions exercise 141 branches, whereas filtering based on FIL_f, FIL_{fp}, and FIL_{dep} results in only 74, 6, and 3 branches to consider.

Interestingly, sets FIL_{fp} and FIL_{dep} are considerably smaller than sets FIL_f, whose size is close to the total number of exercised branches. Moreover, the difference in the sizes tend to decrease as we go down the table, which provides further evidence of the difference in the similarity of passing and failing executions for the different programs. Intuitively, higher similarity would result in a higher number of common branches between passing and failing executions, and thus in smaller FIL_{fp} (and FIL_{dep}) sets.

Table 4: Total number of branches exercised by the synthesized passing and failing executions and size of the corresponding filtering sets (see Section 3.2.2).

Faults	Total	# FIL_f	# FIL_{fp}	# FIL_{dep}
exim	1379	1359	3	3
xmail	141	74	6	3
sed.fault2	715	456	29	11
sed.fault1*	158	130	44	19
grep*	278	276	96	72
aspell*	45	6	0	0
htget*	93	67	0	0
gzip.fault2**	213	202	153	80
socat**	53	36	28	14
rsync**	106	91	57	28

Table 5: Positions of the actual faults in the ranked list produced by the fault localizer when using traditional Ochiai, Ochiai with our three types of filtering, traditional OBM, and OBM with our three types of filtering.

Faults	Ochiai				OBM			
	Orig	FIL$_f$	FIL$_{fp}$	FIL$_{dep}$	Orig	FIL$_f$	FIL$_{fp}$	FIL$_{dep}$
exim	1	1	1	1	1	1	1	1
xmail	1	1	1	1	1	1	1	1
sed.fault2	24	24	24	8	24	24	24	8
sed.fault1*	38	38	38	15	38	38	38	15
grep*	40	40	40	23	48	48	40	23
aspell*	6	6	-	-	6	6	-	-
htget*	67	67	-	-	67	67	-	-
gzip.fault2**	84	84	84	49	84	84	84	49
socat**	26	26	26	13	26	26	26	13
rsync**	11	11	11	6	11	11	11	6

As we explained in Section 3.2.2, when using filtering, F^3 selects a subset of branches and applies fault localization only to the code corresponding to these branches, thus eliminating from consideration a large percentage of program entities. Because of the way the filtering sets are computed, they are not guaranteed to be non-empty for a given program, fault, and set of executions. As Table 4 shows, for the programs we considered, set FIL_{dep} is non-empty in 8 out of 10 cases.

Table 5 shows the results of fault localization enhanced with filtering. Similar to Table 3, the results are presented in terms of position of the actual fault in the ranked list produced by the fault localizer. The columns in the table are divided into two groups, one for each type of fault-localization technique considered, that is, Ochiai and OBM. The table presents four results for each of these two techniques: results computed without filtering (Orig) and results computed with FIL_f, FIL_{fp}, and FIL_{dep} filtering.

The results in the table show that the first two filters, FIL_f and FIL_{fp}, do not improve the fault-localization results (with the only exception of OBM, when run on grep with the FIL_{fp} filter). This result is somehow expected, as entities that are executed by all failing (resp., passing) executions should be assigned high (resp., low) suspiciousness values when using traditional fault localization techniques. Even in these cases, however, filtering based on the FIL_f and FIL_{fp} sets can still be useful because it can considerably reduce the overall size of the ranked list of suspicious entities, as discussed in Section 3.2.2 and shown in Table 4.

The results are different in the case of the third filter, FIL_{dep}, which can considerably improve the effectiveness of fault localization. For the faults in our study, in fact, FIL_{dep} filtering was able to improve the ranking of the faulty entities by 50% on average. Ideally, with these improvements, developers would have to inspect only half of the entities that they would have to inspect using a set of unfiltered results.

Table 6: Positions of the actual faults in the ranked list produced by Ochiai with and without profiling information.

Faults	Ochiai	
	Orig	Profiling
exim	1	1
xmail	1	1
sed.fault2	24	1
sed.fault1*	15	15
grep*	40	38
aspell*	6	1
htget*	67	1
gzip.fault2**	84	3
socat**	26	26
rsync**	11	11

Table 7: Positions of the actual faults in the ranked list produced by the fault localizer when using the original Ochiai and OBM techniques with and without grouping.

Faults	Ochiai		OBM	
	Orig	Grouping	Orig	Grouping
exim	1	1	1	1
xmail	1	1	1	1
sed.fault2	24	17	24	17
sed.fault1*	38	23	38	23
grep*	40	22	48	26
aspell*	6	4	6	4
htget*	67	21	67	21
gzip.fault2**	84	35	84	35
socat**	26	14	26	14
rsync**	11	9	11	9

In general, our results show that filtering can improve traditional fault-localization techniques in two ways: (1) filtering based on the FIL_f and FIL_{fp} sets can almost never improve the ranking of the faulty entity, but it can dramatically reduce the size of the list of suspicious entities; (2) filtering based on the FIL_{dep} set can also improve the ranking of the faulty entities.

Profiling. We now discuss the results of using profiling instead of coverage information for calculating suspiciousness values in Ochiai. As we explained in Section 3.2.3, we considered only Ochiai because OBM cannot be easily extended to use this additional information.

Table 6 shows the results of fault localization enhanced with profiling information. Similar to Table 5, the results in the Orig column are those for Ochiai without any modification, whereas the results in column Profiling are based on the modified Ochiai. As the results in the table shows, using profiling information improves the effectiveness of the Ochiai technique dramatically in 4 out of 10 cases, for sed.fault2, aspell, htget and gzip.fault2, and marginally in one additional case, for grep. (For these cases, using F^3 instead of the traditional Ochiai technique would likely allow developers to locate the fault at once.) In 2 of the other cases, exim and xmail, there is no room for improvement, and for the remaining three cases profiling does not affect the results at all. After manually checking the types of the four faults where profiling information improves effectiveness of fault-localization techniques significantly, we observed that they were mostly buffer overflows, which were caused by bugs inside loops. This finding supports our intuition that profiling is the ideal type of runtime information for localization of some types of faults, as we discuss in Section 3.2.3.

Overall, our results provide evidence that the use of profiling information can dramatically improve the effectiveness of fault localization for at least some types of faults.

Grouping. We now examine the usefulness of our third and last optimization, grouping, when applied to both Ochiai and OBM. To this end, Table 7 shows the comparison between the effectiveness of fault localization with and without grouping.

The results in the table show that also grouping can considerably improve the effectiveness of fault localization. Out of the 10 cases considered, and excluding the 2 cases that cannot be improved because the faulty entity is already ranked first, applying grouping always increases the ranking of the faulty entities. In some cases, such as for aspell and rsync, the improvement is marginal. In most other cases, however, the improvement is considerable.

Answer to RQ3: *For the programs and failures considered, most of our optimizations can actually improve the effectiveness of fault localization, albeit to different extents. Aggressive filtering and grouping can often increase the ranking of faulty entities, and profiling information can dramatically improve the effectiveness of fault localization for at least some types of faults.*

4.4.4 Limitations and Threats to Validity

One of the main limitations of our approach is that it reuses part of the implementation of our previous framework, BugRedux, which in turn relies on symbolic execution—a complex and expensive approach. Despite the inherent technical limitations of symbolic execution, however, recent advances in this area have considerably improved the practical applicability of these approaches.

Like for every empirical evaluation, there are threats to the internal and external validity of our results. First, there may be faults in our implementation that might have affected our results. To address this threat we manually checked many of our results (and did not encounter any error). Another threat is that we used our own implementation of the fault-localization techniques for our studies. Because the techniques considered consist of fairly simple formulas, we feel confident that we implemented the techniques correctly and in an unbiased way. Finally, in our studies we considered only ten programs, so our findings may not generalize to other programs or faults. However, the programs we considered are from different repositories, were used also in previous research [8,15,23], and seven of the faults in these programs are *real bugs found by real users in the field*, that is, *real field failures*. We nevertheless plan to perform an additional extensive empirical evaluation, including user studies, to confirm our results.

5. RELATED WORK

In the past few years, researchers have defined numerous debugging techniques. Among those, we discuss the techniques most closely related to our work.

Statistical fault-localization techniques are a set of techniques that identify suspicious statements by leveraging a large number of passing and failing executions. These techniques share a similar intuition—entities that executed mostly in failing executions are more likely the potential causes of failures. Tarantula [24], uses statement coverage as criteria to compute suspiciousness values and rank them. CBI and followup work from Liblit and colleagues [26, 27] use predicate outcomes, instead of statements, as criteria. Moreover, they operate on data collected from the field, rather than in-house. In recent years, many different statistical fault-localization approaches were proposed that operate on different entities and/or use different statistical analyses to compute suspiciousness values [4–6, 28, 34, 42]. Among these techniques, Ochiai [4] has shown to perform better than other similar techniques in several empirical studies, and recent research has shown, both empirically and analytically, that even better techniques can be defined [30]. Also recently, Baah and colleagues defined a technique [9] that accounts for the confounding bias due to program dependences when computing suspiciousness values and can improve the effectiveness of fault localization. Usually, statistical fault-localization techniques are limited by the fact of requiring a high quality test suite containing a large number of passing and failing inputs. F^3 leverages and extends these techniques (Ochiai and OBM, specifically) and addresses their limitations by generating the needed inputs.

Other researchers defined approaches that extend traditional fault localization techniques and/or try to address their limitations. To enable fault localization in the absence of multiple failing test cases, BugEx [33] uses an evolutionary approach that (1) aims to generate executions almost identical to a single failing test and (2) uses these executions to identify program facts that are relevant for the failure. Although the technique is potentially effective, generating executions that have minimal differences with one another (*e.g.,* only one branch) is extremely challenging, which limits the practical applicability of the approach. Artzi and colleagues use a specialized dynamic symbolic execution approach to (1) discover faults in web applications, (2) generate passing and failing executions similar to the executions that revealed the faults, and (3) use these executions to try to localize the faults [7]. Their technique is not meant to help the debugging of field failures and is specialized for web applications and the faults in such applications (*e.g.,* malformed HTML). Groce [20] propose an idea similar to our filtering approach. Similar to our approach, the effectiveness of their filters highly depends on the quality of the available test suite.

Another family of approaches to fault localization is experimental debugging. Among these techniques, one of the most popular is delta debugging [16], which can simplify and isolate failure causes through a differential analysis of inputs [40], code [38], or program states [39]. More recently, researchers have defined related techniques that perform fault localization by modifying the state of a program in several points during a failing execution and assessing whether the state change prevented the failure from occurring; if so, the corresponding point in the code is reported as potentially faulty (*e.g.,* [22,41]). The main limitation of these approaches is that they are unsound, as the manipulation of program states (and executions in general) can result in infeasible behavior and can thus produce false positives.

Formula based fault-localization techniques perform static and dynamic analyses to transform programs and program executions into formulas and manipulate these formulas to locate suspicious statements. BugAssist [25] transforms a program, an input that makes the program fail, and the negation of the failing assertion for that input into a formula that consists of clauses in conjunctive normal form. It then treats the satisfiability of the formula as a MAX-SAT problem [10], so as to identify a subset of clauses that, if removed, make the formula satisfiable. Finally, it reports the code corresponding to these clauses as a potential cause of the failure. A similar approach, Error Invariants [19], transforms a single failing trace, rather than a complete program, into a formula. This technique leverages interpolants to identify the points in the failing trace where the state is modified in a way that affects the final outcome of the execution. It then reports the statements in these points as potential causes of the failure. One limitation of this technique is that it cannot handle control-flow related faults because it does not encode control-flow information in the formula. To address this issue, in followup work, the authors developed an improved version of the approach that can encode partial control-flow information [13]. Formula based approaches can identify potential causes precisely and in a principled way, but they rely on an extremely expensive analysis that prevents these techniques from being applicable in practice, at least in their current formulation.

6. CONCLUSION AND FUTURE WORK

Understanding and debugging field failures is a notoriously difficult task because of the increasing complexity of modern software systems and the limited availability of information from the field (typically limited to crash stacks). To mitigate this problem, and better help developers identify the likely causes of field failures, we have proposed F^3, a technique that extends our previous work on field-failure reproduction with automated fault-localization capabilities. Given a field failure, F^3 (1) synthesizes passing and failing executions that are similar to the original failure and (2) leverages the generated executions to perform fault localization using a set of suitably optimized fault-localization techniques.

To assess the effectiveness of our approach, we implemented it in a prototype tool and performed an empirical evaluation on a set of *real-world programs and real field failures*. Our results, albeit still preliminary, are promising. First, they show that F^3 was able to synthesize multiple passing and failing executions and successfully use the synthesized executions for fault localization. Second, the results also show that the optimizations that we propose are effective; in many cases, our optimizations were able to (1) reduce the length of the lists of suspicious program entities reported to developers and (2) improve the ranks of the faulty program entities in such lists. Although we must perform user studies to confirm our findings, both of these improvements have the potential to considerably reduce the developers' effort needed to identify the causes of field failures.

Besides performing additional experimentation, our immediate plans for future work involve leveraging the metrics proposed by Sumner and colleagues [35] to measure the degree of similarity within the sets of executions that F^3 generates. We will then look for correlations between similarity within an execution set and effectiveness of fault localization performed using that set. We believe that our findings will help us further improve both our execution synthesis and our fault-localization techniques.

We will also investigate the use of fault-localization techniques other than statistical fault localization, such as formula-based techniques (*e.g.,* [19,25]), and assess whether they can be used to provide an actual explanation of the failure and its causes.

Recently, information theory has been applied to help improve the efficiency of fault localization (*e.g.,* [36]). We will investigate the use of these approaches to guide input generation towards passing and failing executions with large information gain with respect to the fault location. This may reduce the number of passing and failing executions required to locate a fault and ultimately improve the efficiency of our technique (for which input generation is by far the most expensive part).

Finally, call sequences are only one possible type of execution data that we can collect from the field, and symbolic execution is just one possible input-generation technique. We will investigate whether other types of field data and (possibly less expensive) input-generation techniques can be successfully used in the context of our approach.

7. ACKNOWLEDGMENTS

This work was partially supported by NSF awards CCF-1161821, CCF-0964647, and CCF-0916605 to Georgia Tech, and by funding from IBM Research and Microsoft Research. The anonymous reviewers provided valuable feedback that helped us improve the paper.

8. REFERENCES

[1] Software-artifact Infrastructure Repository. `http://sir.unl.edu/`, Apr. 2012.

[2] Technical Note TN2123: CrashReporter. `http://developer.apple.com/technotes/tn2004/tn2123.html`, Apr. 2012.

[3] Windows Error Reporting: Getting Started. `http://www.microsoft.com/whdc/maintain/StartWER.mspx`, 2012.

[4] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proc. of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, 2006.

[5] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An Observation-based Model for Fault Localization. In *Proc. of the 2008 International Workshop on Dynamic Analysis*, pages 64–70, 2008.

[6] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed Test Generation for Effective Fault Localization. In *Proc. of the 19th International Symposium on Software Testing and Analysis*, pages 49–60, 2010.

[7] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Fault Localization for Dynamic Web Applications. *IEEE Transactions on Software Engineering*, 38(2):314 –335, March-April 2012.

[8] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proc. of the 18th Network and Distributed System Security Symposium*, Feb. 2011.

[9] G. K. Baah, A. Podgurski, and M. J. Harrold. Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization. In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 146–156, 2011.

[10] J. Bailey and P. J. Stuckey. Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization. In *Proc. of the 7th International Conference on Practical Aspects of Declarative Languages*, pages 174–186, 2005.

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

[12] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proc. of the 31st International Conference on Software Engineering*, pages 34–44, 2009.

[13] J. Christ, E. Ermis, M. Schäf, and T. Wies. Flow-sensitive Fault Localization. In *Proc. of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2013.

[14] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *ICSE 2007*, pages 261–270, 2007.

[15] J. Clause and A. Orso. PENUMBRA: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting. In *Proc. of the 2009 International Symposium on Software Testing and Analysis*, pages 249–260, 2009.

[16] H. Cleve and A. Zeller. Locating Causes of Program Failures. In *Proc. of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[17] S. Elbaum and M. Diep. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[18] W. Eric Wong, V. Debroy, and B. Choi. A Family of Code Coverage-based Heuristics for Effective Fault Localization. *Journal of System and Software*, 83(2):188–208, Feb. 2010.

[19] E. Ermis, M. Schäf, and T. Wies. Error Invariants. In *Proc. of the 18th International Symposium on Formal Methods*, pages 187–201, 2012.

[20] A. Groce. Error Explanation with Distance Metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.

[21] D. M. Hilbert and D. F. Redmiles. Extracting Usability Information from User Interface Events. *ACM Computing Surveys*, 32(4):384–421, Dec. 2000.

[22] D. Jeffrey, N. Gupta, and R. Gupta. Fault Localization Using Value Replacement. In *Proc. of the 2008 International Symposium on Software Testing and Analysis*, pages 167–178, 2008.

[23] W. Jin and A. Orso. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proc. of the 34th International Conference on Software Engineering*, pages 474–484, 2012.

[24] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proc. of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[25] M. Jose and R. Majumdar. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In *Proc. of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–446, 2011.

[26] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

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

[28] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical Model-based Bug Localization. In *Proc. of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 286–295, 2005.

[29] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[30] L. Naish, H. J. Lee, and K. Ramamohanarao. A Model for Spectra-based Software Diagnosis. *ACM Transactions on Software Engineering Methodology*, 20(3):11:1–11:32, August 2011.

[31] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *Proc. of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209, July 2011.

[32] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proc. of the 21st International Conference on Software Engineering*, pages 277–284, 1999.

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

[34] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight Fault-localization Using Multiple Coverage Types. In *Proc. of the 31st International Conference on Software Engineering*, pages 56–66, 2009.

[35] W. N. Sumner, T. Bao, and X. Zhang. Selecting Peers for Execution Comparison. In *Proc. of the 2011 International Symposium on Software Testing and Analysis*, pages 309–319, 2011.

[36] S. Yoo, M. Harman, and D. Clark. Fault Localization Prioritization: Comparing Information Theoretic and Coverage Based Approaches. *ACM Transactions on Software Engineering Methodology, to appear*.

[37] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proc. of the 5th European Conference on Computer Systems*, pages 321–334, 2010.

[38] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conference Held Jointly With the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–267, 1999.

[39] A. Zeller. Isolating Cause-effect Chains From Computer Programs. In *Proc. of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, 2002.

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

[41] X. Zhang, N. Gupta, and R. Gupta. Locating Faults Through Automated Predicate Switching. In *Proc. of the 28th International Conference on Software Engineering*, pages 272–281, 2006.

[42] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing Propagation of Infected Program States. In *The 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 43–52, 2009.

[43] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*, 36(5):618–643, Sept. 2010.