Using Likely Invariants for Automated Software Fault Localization

Swarup Kumar Sahoo   John Criswell   Chase Geigle   Vikram Adve

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois USA
{ssahoo2,criswell,geigle1,vadve}@illinois.edu

Abstract

We propose an automatic diagnosis technique for isolating the root cause(s) of software failures. We use likely program invariants, automatically generated using correct inputs that are close to the fault-triggering input, to select a set of candidate program locations which are possible root causes. We then trim the set of candidate root causes using software-implemented dynamic backwards slicing, plus two new filtering heuristics: dependence filtering, and filtering via multiple failing inputs that are also close to the failing input. Experimental results on reported software bugs of three large open-source servers show that we are able to narrow down the number of candidate bug locations to between 5 and 17 program expressions, even in programs that are hundreds of thousands of lines long.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids; B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault-Tolerance

General Terms Algorithms, Performance, Reliability

Keywords Software reliability, Root cause, Testing, Debugging, Fault localization, Invariants, Bug Diagnosis, Program analysis

1. Introduction

Software bugs are *everywhere*. Not only do they infest software during development, but they escape our extermination efforts and enter production code. In addition to severe frustration to customers, software failures result in billions of dollars of lost revenue to service providers [4, 11, 38].

Given a program, *bug detection* is the problem of determining whether the program has bugs, preferably with test cases that trigger such bugs. Bug detection tools such as Valgrind [34], FindBugs [1], Fortify [2], Coverity [3], and many others are widely used in software development today.

Given a program and a "failing input," i.e., an input for which the program encounters a failure, *bug diagnosis*, or "debugging," is the problem of identifying the *reasons*, including the location, cause, and possible fixes for the failure. A key component of bug diagnosis is *fault localization*, the problem of identifying one or more possible locations in the program where code needs to be changed to prevent the failure. Fault localization is an important and time-consuming step in debugging software failures. Currently, bug diagnosis is mostly a manual process, either during development or, during production-run failures. The latter are especially difficult to diagnose because it is difficult to reproduce the failures at the developers' site and because privacy and economic concerns severely limit what information is available from the end-user's site [20]. Automating this diagnosis process can significantly increase programmer productivity and reduce software development costs.

There has been much previous work on this problem [5, 10, 12, 16, 24, 27, 28, 30, 41, 42, 44], but the overall problem is far from solved, and, to our knowledge, *no automated fault localization or bug diagnosis tools are used in real world development*. Simple statistical techniques such as Tarantula [26, 27] and Ochiai [5] turn out to be useful only in relatively few bugs, as our results show (see Section 8.3). Some tools [45] are slow and may not be suitable for large applications or for production-site diagnosis systems. Other tools [16, 24, 35, 41] may report too many source lines as potential root causes of the bug (i.e., report too many false positives) or often fail to report source lines containing the root cause (i.e., often have false negatives). Some tools [16] also may be impractical due to special hardware requirements. Others use systematic but unscalable techniques to compare the memory states of passing and failing runs [41]. In fact, for many of these approaches, it is not clear how they will scale to large, realistic programs.

Likely program invariants are a powerful tool for detecting and diagnosing software errors [18, 24, 35]. Informally, *likely program invariants* are program properties that are *observed* to hold in some set of successful executions but, unlike true invariants, may not necessarily hold for all possible executions.

By training likely program invariants on "good inputs," violations of these invariants in a failing run can identify deviations between good and bad runs (i.e., possible bug locations and their consequences). The tradeoff, however, is that they can also be violated on valid behaviors not captured in the training runs ("false positives") and, if the invariants are too broad, can miss bug locations ("false negatives"). Despite these tradeoffs, such invariants are powerful because they can capture properties *of the actual source code*, including properties that violate programmer intent (whereas programmer-written invariants or assertions in the code, will only express programmer intent), and, by observing patterns of behavior of program variables, (e.g., "the variable `month` lies between 1 and 12"), they can capture deep algorithmic properties which are captured in limited ways, if at all, by static analysis [17].

We propose a sophisticated approach combining likely program invariants with novel filtering techniques to narrow down a possible

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03...$15.00

set of locations that have to be changed to eliminate a failure ("*candidate root causes*"). Our goal is to make this set as small as possible. We improve over previous work in two key ways.

First, all the previous invariant-based approaches except one [24] use general test inputs for training invariants, which has two serious disadvantages. One, using general test inputs makes the likely invariants very broad (e.g., observed value ranges of variables may be large) and not tailored to the specific failure, which can make them miss root causes. Two, because the code coverage from test inputs is often not high [32], root causes from the untested parts of the code can be missed (and the untested parts of the code are also likely to contain bugs). Our first key improvement over previous work is that we generate likely invariants by using a small set of training inputs *that are as close to the given failing input as possible*. Any of these invariants that fail during the failing run capture differences between the failing and passing runs. In this sense, our invariants can be seen as a way to isolate the differences between failing and similar passing runs, which is the original insight behind *Delta Debugging* [44], but the invariants in our approach are a much more lightweight (though less precise) method for comparing program states than the explicit state comparisons used in Delta Debugging. The invariant failures give us a set of program locations that may be candidate root causes of the failure.

Our second key improvement is that we use a more sophisticated series of filters, including two novel techniques, to filter out false positives. Filtering is important due to how our system creates invariants. *We consider it important not to miss the root cause, and instead prefer to generate many candidate root causes and then use sophisticated filtering techniques to eliminate false positives.* We therefore use a very *small* set of training inputs (e.g., only 8 inputs for each bug in our experiments, compared with hundreds or thousands in traditional approaches [18]), which yields much narrower invariants. Narrower invariants are more likely to be violated (leading to fewer false negatives), but many other spurious invariants may be violated as well (leaving a large number of false positives).

We therefore use a combination of novel and known techniques to eliminate the false positives from the initial candidates. We first use dynamic backward slicing to filter out invariants that do not influence the observed failure symptom. We then use two novel heuristics to further narrow down the root causes of failure. The heuristics are based on two observations. First, if an invariant on one instruction fails, then another instruction dependent on the first one may also be likely to have an invariant failure, but the underlying cause is the first, not the second. We remove the second from the candidate root causes and call this the *dependence filtering* step. For efficiency, this heuristic leverages the dynamic backward slicing directly. Second, if multiple similar inputs produce the same failure symptom, they are likely to have the same root cause, and therefore, we can focus on invariants that fail for all such inputs. Again, this heuristic directly leverages (indeed, simply reapplies) the previous steps, but for additional failing inputs.

In summary, the two key insights underlying our work are:

1. Generating likely invariants using training inputs close to the failing input is less likely to miss root causes than using invariants generated from a large number of unrelated inputs (e.g., test inputs); and
2. Using sophisticated filtering techniques can allow us to start with a large number of initial candidate root causes (reducing false negatives), and narrow them down to a small set of final locations (reducing false positives).

We evaluate our techniques experimentally using three large open-source servers: Squid, Apache web server and MySQL. These programs are far larger than those used in any previous study we know of except one [42]. We used eight previously reported bugs, including four memory corruption and four incorrect output bugs. We use an automated procedure to construct the passing and failing inputs that are "close" to the failing input. The results are extremely strong. Overall, we reduce the number of candidate root causes to a range of only 5–17 program expressions per bug, even in programs of hundreds of thousands of lines of code.[1] Moreover, each of the filtering steps proves important: slicing removes nearly 80% of false candidates, dependence filtering removes nearly 58% of the remaining false positives, and using multiple faulty inputs removes several more. Comparing our work against the Tarantula [26, 27] and Ochiai [5] approaches, which have been used for similar comparisons in previous work, we find that their approach is slightly better in the two best cases, but is extremely inconsistent, with thousands of candidate locations in 6 out of 8 cases.

To summarize, our main contributions are:

1. We present a novel invariant-based approach for fault localization, based on training the invariants with a small set of automatically constructed "good inputs" that are close to a given failing input. Our results show that this approach yields a more precise set of root causes than previous work.
2. We present novel heuristics for reducing false positives in the diagnosis results. These heuristics have valuable synergy with our core invariant strategy because they leverage the other analysis techniques, including input construction and dynamic backward slicing.
3. We evaluate our approach for real bugs in a larger and more realistic set of applications than all previous work except one.
4. Our results show that the approach is extremely effective at reducing the set of possible root causes, even in large programs, and that each of the filtering steps makes an important contribution in reducing this set.

The rest of the paper is organized as follows. Section 2 motivates and gives an overview of our invariant-based diagnosis framework. Section 3 describes the design of our invariant-based analysis. Sections 4 and 5 describe techniques that prune false positives. Section 6 briefly describes how the results of the diagnosis is presented to the programmer. Section 7 describes our experimental methodology and applications. Our experimental results are presented in Section 8. Related work is discussed in Section 9, and we describe future work and conclude in Section 10.

2. Overview of Our Approach

Some previous automated bug diagnosis techniques use program invariants [24, 30]. Others, such as Delta Debugging [12, 44], compare a failed execution of the program with a successful execution to identify the root causes of a failure. One drawback of Delta Debugging is that comparing individual dynamic statements or values between a failed and a successful run can be inefficient and unscalable. Instead, we can use program invariants to make an efficient comparison of the program runs. Thus, *our approach essentially combines these two prior approaches*.

Our approach needs the source code of the program, configuration files, a failing input and a deterministic detector (described in more detail in Section 7.2). Figure 1 gives a broad overview of our approach. The process begins with a program and a failing input, plus an optional grammar or *specification* specifying the possible tokens in valid program inputs (in the form of a lexical analyzer or tokenizer) and all possible replacement tokens for each input to-

[1] Although total program size is not a good metric of programmer effort in narrowing down a bug, it is a good metric for the difficulty faced by automated tools which must begin with the entire program.

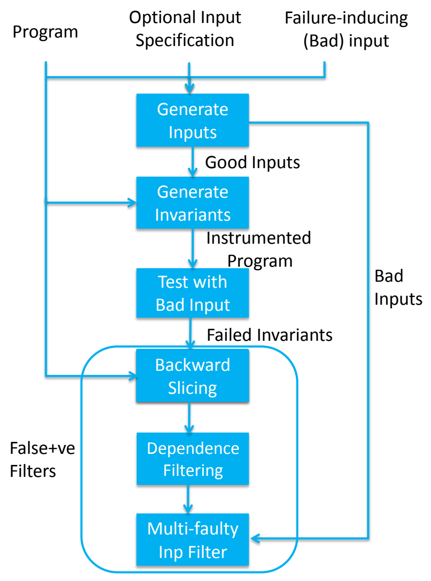

Figure 1. Diagnosis Tool Architecture

ken. The output is a set of source-level program expressions that identify locations of the possible root causes. Note that our current approach only uses the specification of tokens and their replacements, if available; it does not need any production rules of the input grammar. If this information about the grammar is not available, our approach falls back on an application-oblivious character-level rewriting algorithm, very similar to the tool, `ddmin` [45]. (A final step mapping invariants back to source program expressions is not shown.)

Program invariants are predicates that are guaranteed to hold for all possible inputs. Predicates observed to hold for some, but perhaps not all, program inputs are known as *likely invariants* [18]. Likely invariants can be extracted by monitoring the execution of the program for some desired set of program inputs; we refer to these as the *training runs*. If we extract likely invariants from some set of training runs of a program, instrument the program to test the likely invariants, and then execute it with the failing input, any invariants that fail are a strong indicator of differences from successful runs. Thus, we can use locations of failed invariants as an initial set of *candidate root causes* of the bug.

One key problem we must address in this work is that this initial set of candidate root causes is often very large, e.g., with hundreds of locations. We refer to all except the true locations of the root cause as *false positives*. The challenge is to filter down the set of candidates into as small a set of locations as possible. We use a series of dynamic analysis steps based on dynamic program slicing and dependence filtering to filter out as many false positives as possible.

We use the example in Listing 1 to illustrate the steps of our approach. This is a simplified version of some code in the MySQL database server which contains a buffer overflow bug. The buffer overflow occurs when a *Select* query contains a specific date between 0000-Jan-01 and 0000-Feb-28. The overflow occurs at line 31, when the `type_names` array is indexed with `weekday` for the aforementioned dates. The root cause of this bug is at line 10 in function `calc_daynr`. It uses an unsigned variable `year` to perform computation. When the value of `year` is 0, the decrement statement at line 10 results in a very large number and, consequently, the value of `temp` at line 13 also becomes a very large number. Finally, at line 14, the function returns a negative number; this value is used to calculate the day of the week in function `calc_weekday`, which, in turn, returns a negative value. Hence, when the return value is used as an index in line 31, it causes a buffer overflow and a program crash.

2.1 Likely Program Invariants

We use program invariants to find a set of candidate program instructions that are possible root causes of a failure. Given an input that triggers the bug, we derive a set of good inputs that are close to the failure inducing input in lexicographic distance, and do not trigger the bug. We do this automatically using a variant of the *ddmin* algorithm [45], using character-level or token-level rewriting depending on whether the specification (grammar tokens and replacements) are known. We then run the program with these good inputs to find a set of program invariants. The invariants we use are *range invariants* (described in Section 3.1.1). We limit our invariants to load, store or function return values, for reasons explained in Section 3.1. For example, we monitor the return value of functions such as `calc_daynr` and `calc_weekday`, as well as the loads of the three fields of `t` in the arguments to `calc_daynr`. For all good inputs, the return values of the two functions will be positive; this will be captured as likely invariants. When the program is run on the failure-inducing input, these two invariants (along with some other invariants) will fail as the return values of these functions will be negative. This will give us a number of candidate locations which can be root causes.

Overall, this step generated 95 failed invariants for this particular bug in MySQL. While this is far too large to report to programmers, it is still an impressively low number (especially considering that we are using one of the simplest possible types of invariants, ordinary ranges). This is strong evidence that the overall approach based on invariants constructed from nearby good inputs is promising.

2.2 Dynamic Program Slicing

Most failed invariants are not root causes of the bug. For example, some failed invariants are on instructions that do not compute any values that affect (directly or transitively) the instruction at which the symptom occurs. We can eliminate any such instruction from the set of possible root causes.

Dynamic backwards slicing [48] is a technique that can precisely determine which instructions affected a particular value in a single execution of a program. We start from the statement which causes the symptom (in this example, the invalid array element access) and find the statements in the execution that could have affected that statement, either via data flow or control flow. This set of statements is exactly the dynamic backward slice. We then remove all the failed invariants which are not part of this slice from the set of possible root causes.

In the part of the code shown in Listing 1, there are actually no failed invariants for the failing execution (though there are fairly many in other parts of the program). However, imagine that statements 25 and 26 had generated failed invariants. They would have been false positives since they are not the root cause of the failure. However, these statements are not part of the dynamic backward slice because they do not affect the value at the symptom, and so they would have been filtered out from the candidate set. Note that both function returns are part of the dynamic backward slice and would be included in the candidate set.

For this bug, this step removes about 62% of the failed invariants, reducing the candidate count from 95 to 36.

2.3 Dependence Filtering

Another source of false positives is error propagation. If a faulty instruction that is the bug's root cause triggers an invariant fail-

```
long calc_daynr(uint year,uint month,uint day)
{
  long delsum;
  int temp;

  if (year == 0 && month == 0 && day == 0)
    return (0); /* Skip errors */
  delsum= (long)(365L*year+31*(month-1)+day);
  if (month <= 2)
      year--;
  else
    delsum-= (long) (month*4+23)/10;
  temp=(int) ((year/100+1)*3)/4;
  return (delsum+(int) year/4-temp);
}

int calc_weekday(long daynr, bool first_week_day)
{
  return ((int)((daynr+5L+(first_week_day ? 1L : 0L)) % 7));
}

bool make_date_time(TIME_FORMAT *format, M_TIME *t,
             timestamp_type type, String *str)
{
   str->length(0);
   if (l_time->neg)
     str->append('-');
...
  weekday= calc_weekday(calc_daynr(t->year,t->month,
                          t->day),0);
  str->append(loc->d_names->type_names[weekday],
      strlen(loc->d_names->type_names[weekday]),
                          system_charset_info);
...
}
```

Listing 1. Source Code Example

ure, then any instruction using the faulty value computed by that instruction may also trigger an invariant failure. Such subsequent invariant failures are not indicative of a root cause; they merely occur because the erroneous value is propagating through subsequent computation.

Using this observation, we have devised a heuristic called *dependence filtering* to further reduce the set of candidate root causes. This heuristic uses the data flow graph and control dependence graph formed during the previous dynamic slicing step to eliminate any instruction with a failed invariant that depends on another instruction with a failed invariant, *with no intervening passing invariants*. This policy is explained in Section 5.1.

For the example bug, when we build the dynamic dependence graph starting from the failing array indexing statement, the failed invariants of `calc_daynr` and `calc_weekday` will lie on one path to the symptom. Since the result of `calc_daynr` is directly used by `calc_weekday` with no intervening instructions that have invariants, the error at `calc_weekday` is likely to be propagated from `calc_daynr`, and we exclude the return value of `calc_weekday` as a candidate.

This step removes about 55% of the 36 remaining candidate root causes, leaving only about 16 candidates.

2.4 Filtering Using Multiple Failing Inputs

Note that once the invariants have been computed, the remaining steps above (finding failing invariants; dynamic backward slicing; dependence filtering) all happen using a single failing input. However, we can construct multiple inputs similar to the original input that also cause the program to fail with exactly the same symptom. (We do this using the same variant of the ddmin algorithm.) Since these new bad inputs result in the same failure, it is very likely that the root cause will be the same. We can therefore *repeat the last three steps above* for each such failing input. The root cause should be included in the final candidates generated for all these inputs. We can then take the intersection of these sets to find the final set of candidates. For this bug, this step reduced the final set of candidates by 25% from 16 to 12.

The tool will first present the set of candidate locations from the final step to the developer for analysis. If the developer cannot find the root cause among them, then the tool will present the other candidate root causes in the order depending upon the number of other failing inputs in which they appear. If the root cause cannot still be identified, then the tool will present root causes from previous steps in this order - dependence filtering, slicing and likely invariants.

Overall, our diagnosis has narrowed down the possible location of the bug *to 16 locations in a program of over 1 million total lines of code* for this bug since the root cause of this bug cannot be found in the final step but can be found among the candidate locations after the dependence filtering step. Moreover, for each of these locations, we have fairly detailed information about the ranges of program values seen in the successful executions and the corresponding actual values from the failing runs which fall outside those ranges. This is an extremely effective result because it should vastly simplify the task of the programmer in debugging the error.

2.5 Feedback to the Programmer

Each candidate invariant in the final or non-final set represents an instruction that may indicate the location of a root cause of the failure. Recall, however, that we only track invariants on load, store and return instructions. This means that the actual location of a bug may be anywhere in the maximal sub-expression rooted at a load, store or return that does not itself contain any load, store or return. (We call this a *pure, local sub-expression*). Therefore, the tool additionally computes the maximal pure, local sub-expression rooted at each candidate root cause and presents it back to the programmer.

For the example bug, the invariant on the return value of `calc_daynr` indicates the location of the root cause, but the true location is somewhere in the body of that function. (Note that the entire function body is pure and local, after register promotion of local stack variables.) Therefore, the tool would present to the programmer the subset of the body of the function that is used to compute the return value in the failing run, i.e., lines 6, 8, 9, 10, 13 and 14. Of course, the negative return value in the failing run is a huge clue that `temp` at line 14 must be a large value. `temp` is computed from `year`, and `year` is computed by decrementing the incoming parameter value. This can only produce a large value by overflowing. Thus, the decrement of `year` is the true root cause.

2.6 Program Analysis Infrastructure

Our system uses the LLVM Compiler Infrastructure [29] to analyze and transform programs. LLVM supports multiple languages, including C and C++. Its intermediate representation (the LLVM IR) represents a program as a set of functions; each function is a set of basic blocks with an explicit, statically known control-flow graph. Scalar variables are kept in Static Single Assignment (SSA) form [13], making definition-use chains explicit. Other variables (e.g., C arrays and structs) are kept in non-SSA form; scalar fields of structures and array elements are accessed using RISC-like load and store instructions.

3. Diagnosis with Invariants

We make some assumptions about the inputs and the programs being diagnosed. First, we assume that the inputs to the program can be described by some formal *specification* (specifying the possible tokens in valid program inputs and all possible replacement tokens) suitable for driving an input generation algorithm based on the *ddmin* approach for minimizing failure-inducing inputs [45]. This specification, however, is optional. If the specification is not available, we can use our default specification-independent character-level rewriting approach for input generation. Second, we assume that the program includes a set of *deterministic detectors* to detect failures. For example, memory safety errors can manifest in non-deterministic ways, but we can use detectors like SAFECode [14], Valgrind [34] or SoftBound [33] to ensure that they provide the same symptom every time they are triggered. These detectors are only used offline after a failure is detected; the detectors are not used, and therefore add no overhead, in production runs of the program. For incorrect output errors, we assume that programmer-inserted assertions can be used for detection (these are often available at development time) or “known outputs” are available (e.g. from nightly testing, different version of the application) for failure detection. Also, these detectors can be used online during development time, since overhead is not an issue during development time. Failure detectors are further discussed in Section 7.2.

The input to our diagnosis is a program, required configuration files, the “failing input” that triggered the failure, and the failure symptom detected by some deterministic detector.

3.1 Constructing Candidate Root Causes

When a program fails on a particular input but succeeds for an input that is very similar (i.e., close) to the failing input, there are likely to be relatively few differences in behavior between those two executions; the key insight behind the Delta Debugging work is that these differences can help narrow down the root cause(s) of the failure. Our goal is to perform this narrowing automatically. If we can construct a set of such good inputs, then we can automatically extract behavior patterns that are common to these inputs. If the execution with the failing input violates any of these behavior patterns, that violation is a powerful indicator of the possible root cause(s) of the failure.

3.1.1 Likely Range Invariants

In this work, we use a simple form of program invariants — likely range invariants. A *likely range invariant*, or range invariant for short, is a range of values computed by some program instruction in the monitored runs. For example, if an instruction `load int* %p` is always observed to return a value in the range between -5 and 10 (denoted [-5,+10]) in some set of executions, then that range represents a range invariant for that instruction. A range invariant can be *tested* in future runs by instrumenting the program to compare the value computed by the instruction to see whether or not it falls within the range.

Range invariants are attractive because they capture concisely the observed values of program variables and instructions. They are also efficient to generate and monitor. The number of such invariants is linear in the size of the program (unlike e.g., invariants comparing pairs of variables). Although other data and control invariants are likely to be useful in identifying candidate root causes, we leave it to future work to explore such choices.

Since programs can have many instructions, we limit our monitoring of invariants to load, store, and function return instructions. For example, an assignment statement, `A[i] = a + b * B[j];` may have an error anywhere in the expression, but we only monitor the load of `B[j]` and the store to `A[i]` (there are no loads/stores of a and b as they are register promoted locals). This means we cannot narrow down the location of an error to anything smaller than an entire subexpression terminating in a load or store, but we expect this granularity to be ample for programmers to pinpoint the root causes of a failure. It may not be obvious why we monitor loads as well as stores when loads almost always return values from previous stores. We do so because it allows us to detect errors in array index expressions, such as `A[i]` or `B[j]` above.

We extract range invariants from training runs using the good inputs generated from the original failing input. We generate invariants only for instructions that are executed by at least one of the good inputs. We then instrument the program to test those invariants and re-execute the instrumented program using the failing input. If a particular range invariant is violated, i.e., the relevant instruction produced a value outside the observed range, we assume that the relevant instruction may contain the bug, i.e., we consider it to be a candidate root cause of the failure.

3.1.2 False Positives

The technique used to extract likely invariants can affect the accuracy and the number of candidate root causes. When a large number of arbitrary program inputs are used to extract range invariants (e.g., all inputs used during software testing), the ranges obtained are likely to be wide. *This makes it less likely that such a range invariant will be violated during a particular failing run,* which means that the root cause of a failure may not be included in the initial set of candidate root causes. This was the approach taken by previous systems [16, 24].

Instead, we construct “good” inputs that are (a) result in execution whose control flow and intermediate program values are as close to that of the failing run as possible and, (b) few in number, and use those to extract range invariants. Using such good inputs has two benefits. First, it ensures that the behavior of the training runs will be as close to the failing run as possible, and so tends to isolate the differences that might be the causes of the failure. Moreover, it allows us to use a very small set of training inputs (e.g., only 8 as opposed to thousands, as used in prior work [18]). For example, consider a bug in MySQL that is triggered by an uncommon `Date` field in a Query *SELECT DATE_FORMAT(”0000-01-01”,’%W %d %M %Y’) as a;*. Constructing invariants using a general set of inputs would include queries for INSERT, DELETE, SELECT, JOIN, etc., most of which are likely to have very different behaviors from the failing input. One would need a large number of such inputs to be representative of program behavior. Using a small number of inputs that are variations of the previous query with different `Date` fields that do not trigger the error requires a much smaller set of inputs to represent the behavior we are seeking.

Using only a few such good inputs makes the invariants narrower (e.g., smaller ranges for range invariants), which is both good and bad. It is good because it makes it more likely that the different behavior will result in a failed invariant. It is bad because it may cause a large number of invariants to fail. Since we aim to identify the root cause(s) of a failure, it is important not to miss the root cause and so the benefit far outweighs the disadvantage. One key goal of this work, therefore, is to compensate for the disadvantage, i.e., to filter out the false positives, reducing the final set of reported candidates as much as possible without eliminating the true root causes. Filtering techniques we employ are described in detail in Section 5.

As explained earlier, success of our approach depends upon generating good inputs with similar execution behavior like program control flow and intermediate values as that of the failing input and training invariants using those inputs. Hence, the first step of our overall process is to construct such a set of “good” inputs which is discussed in next Section.

3.2 Input Construction

Algorithm 1 Deletion-based Input Generation

```
function INP-GEN-DELETE(input, num)
    GoodInputs ← a set of good inputs from ddmin algorithm
    return good-inp-gen (GoodInputs, num)
end function

function GOOD-INP-GEN(inputs, num)
    Initialize queue GoodInpQ with inputs
    while GoodInpQ ≠ ∅ do
        goodInput ← removeFront (GoodInpQ)
        add goodInput to GoodInputList
        find-good-inp (goodInput, length (goodInput))
    end while
    sort GoodInputList based on edit-distance
    return first num elements of GoodInputList
end function

function FIND-GOOD-INP(input, n)
    Split input into n subsets of equal size
    for each subset s do
        if s is a good input then
            enqueue(GoodInputQ, s)
        end if
        if complement of s is a good input then
            enqueue(GoodInputQ, complement of s)
        end if
    end for
    if n/2 > 1 then
        find-good-inp (input, n/2)
    end if
end function
```

The first step of our overall process is to create inputs with similar execution behavior by constructing a set of “good” inputs that are as close to the failing input as possible. By “good,” we mean that the inputs do not trigger a failure with any of the detectors in the program.[2]

We have designed two input generation algorithms that construct such similar good inputs by systematically modifying parts of a small failing input (they can also be used to generate additional failure-triggering, or bad, inputs by modifying the criteria they use for selecting which inputs to return). These algorithms use two different approaches to generate inputs–one uses a deletion based approach described as *inp-gen-delete* in Algorithm 1, and the other uses a replacement based approach described as *inp-gen-replace* in Algorithm 2. The second approach is more powerful but requires input specification to perform token-level rewriting. The first approach is a fallback character-level rewriting approach in case a specification is not available. Both *inp-gen-delete* and *inp-gen-replace* take an input (e.g., the input that triggers the bug) and the number of desired good inputs and return a set of good (or failure-inducing) inputs.

The deletion approach initially uses *ddmin* [45] (a systematic procedure for generating minimal inputs that also produces good and bad inputs as a side-effect) to generate a few good inputs. Since, in many cases, *ddmin* produces too few good inputs, we also use *good-inp-gen* (a variation of the original *ddmin* algorithm) to produce more good inputs. This function starts by removing single characters and then removes exponentially more characters from the input as it tries to find more good inputs. Like *ddmin*, this also tries both a substring and its complement to increase the likelihood of quickly finding good inputs. The algorithm can determine whether an input is good or bad by running the application on the input and determining if the input triggers the same fault as the original failure-inducing input. After the good inputs are generated, *good-inp-gen* sorts the inputs in the order based on its edit-distance from the original failure-inducing input and returns a pre-defined number of elements from the front of the list.

Both *ddmin* and *good-inp-gen* do not require knowledge of the input specification, so they are easy to deploy. However, by utilizing the specification, we can create a more powerful input generator. An example of such a generator is *inp-gen-replace* which needs an input specification specifying the possible tokens in valid program inputs and all possible replacement tokens for each input token. This generator uses the specification to identify tokens for each terminal in the input. It then systematically generates many variations for each token depending upon the type of the token (note that, in Algorithm 2, there is a *mk<type>Variations()* function to generate inputs for each type of token). For example, *mkStrVariations()* systematically divides strings into 2^i parts for various values of i and then replaces each character with some other characters, which runs in time proportional to the size of the string. Similarly, *mkIntVariations()* systematically creates different integer values which differ by 2^i from the original value for various values of i. For grammar symbol tokens which are not of any special type, we replace them with their matching tokens depending upon the input specification rules. For example, a token for an aggregate function in MySQL like MAX can be replaced by matching tokens like MIN, AVG, and SUM. Then, we generate inputs by combining the different replacement values for each token. Currently, we generate inputs each of which contain variations of only one token, as exponential blow-up is possible if we consider variations of many tokens in each input.

While building an input generator that utilizes an optional input specification may seem onerous, we believe that it can be made

[2] We expect that the approach would work even if such inputs trigger other unknown failures because our main requirement is to isolate differences that avoid triggering the failure being diagnosed. We have not encountered this situation so far.

Algorithm 2 Replacement-based Input Generation

```
function INP-GEN-REPLACE(input, num)
    GoodInput ← a good input from ddmin or original input
    return inp-replace-input-gen (GoodInput, num)
end function

function INP-REPLACE-INPUT-GEN(input, num)
    GoodInputs ← $\emptyset$
    for each token t in the input do
        if t is string then
            newInputs ← mkStrVariations (input, t)
        else if t is integer then
            newInputs ← mkIntVariations (input, t)
        else if t is float then
            newInputs ← mkFloatVariations (input, t)
            ...
        else if t is grammar symbol then
            newInputs ← mkSymbolVariations (input, t)
        end if

        newGood ← subset of newInputs that are good inputs
        GoodInputs ← GoodInputs $\cup$ newGood
    end for

    sort GoodInputs based on edit-distance
    return first num inputs in GoodInputs
end function
```

practical. We have already implemented it for MySQL, Squid and Apache bugs. First, it should be possible to build "input generator generator" tools (in the same vein as tools like Lex, Yacc and Bison) that semi-automate the creation of input generators. Such a tool would take, as input, the optional specification and a set of functions to generate new strings by replacing individual tokens; the output would be a program that generates inputs for that specification. Second, many programs use standardized input grammars (e.g., all web servers use the HTTP protocol grammar; many databases use the SQL grammar). Once an input generator has been created for a input grammar, it can be reused by many applications using that grammar (since all of them will also use the same specification). Third, creating an input generator is a one-time effort. For large applications that are used over many years, the short-term effort to create an input generator should yield long-term savings in bug diagnosis time. Finally, we currently need only the specification of input tokens (lexical analyzer) and their possible replacements; the individual production rules of the grammar are unnecessary. Programs with complex inputs, like MySQL databases and compilers, have explicit tokenizers which drive their parsers, and it is not be very difficult to use this for our input construction algorithm. The first author implemented this for MySQL. It was not difficult, even though he is not a MySQL developer. In the worst case, we can fall back on the default specification-independent character-level rewriting approach of Algorithm 1 or other fuzzing-based input generation techniques [9, 19].

These algorithms are likely to produce many good inputs since the number of good inputs are likely to much more than failure-inducing inputs. (Both algorithms filter out invalid inputs by examining the output of the execution.) We can also use a variation of these algorithms to construct some failure-inducing bad inputs as well. We select bad inputs instead of good inputs in the appropriate steps to do this.

By constructing inputs close to the original input, and yet avoiding the original failure, these algorithms carefully try to create inputs which minimize the differences between the failing and non-failing executions, thus increasing the likelihood of better diagnostic accuracy. The algorithms presented here are just two possible input generation algorithms; other algorithms may work, too. We also observe that multiple input generators can be used in a single debugging session to provide a variety of inputs.

4. Dynamic Backwards Slicing

We can first discard invariants on values that *do not* affect the behavior of the faulting instruction. We use dynamic backwards slicing starting from the failure symptom to find these values. Dynamic backwards slicing traces a program's execution and then finds the precise set of instructions that directly or indirectly influenced the result of a given value in that execution [48]. For each execution of the program that triggers the failure symptom, we compute the dynamic backward slice of the symptom. Only failing invariants for instructions on this slice are retained. Our dynamic slicing system handles both data-flow and control-flow dependences when computing the dynamic backwards slice. Supporting control-dependence allows our tool to find failed invariants on values that control whether pieces of code relevant to the bug are executed.

Most dynamic backwards slicing systems have two phases [48]: in the first phase, they instrument the code to record, in a trace file at run-time, sufficient information to find the backwards slice of any value. In the second phase, an analysis uses the execution trace to create a program dependence graph that can be used to find which computations affected the result of a value within that specific program execution. Since an execution trace is used, dynamic backwards slicing is precise; it will not include values that did not influence the computation in question [48].

We have implemented a dynamic backwards slicing compiler pass called *Giri*. Giri is very similar to Zhang and Gupta's No Preprocessing with Caching (NPwC) algorithm [48] in that it does not precompute a complete program dependence graph from the trace; instead, it consults the trace only on demand to compute the dynamic backwards slice of a variable and caches the result in memory for subsequent queries.

Giri takes advantage of the LLVM IR to reduce the size of the trace file. LLVM IR represents a program as a set of functions; each function is a set of basic blocks with an explicit, statically known control-flow graph [29]. Scalar variables are kept in Static Single Assignment (SSA) form [13], making the definition-use chains explicit [29]. Other variables (e.g., C arrays and structs) are kept in non-SSA form; scalar struct fields and array elements are accessed using RISC-like load and store instructions [29].

Giri instruments code so that it records three different pieces of information during the first phase: (a) basic block exits; (b) memory accesses and their addresses; and (c) function calls and returns. Using these values and the LLVM program representation (which holds all scalar temporaries in Static Single Assignment (SSA) form), Giri can easily construct the precise dynamic backward slice in the second phase.

Giri's instrumentation works as follows. First, Giri adds an instruction to each basic block that creates a record in the trace when the basic block has finished execution. These trace records are used in the second phase to determine which branches were taken dynamically during the program's execution. Note that recording each basic block's execution (as opposed to recording each individual instruction's execution) suffices since all other instructions in a basic block must have executed before the basic block's last instruction.

Second, Giri instruments the program to record the memory locations accessed by all loads, stores, and select C library functions. During the second phase, when it encounters a load instruction while finding the backwards slice, Giri can use the trace to find the store that created the value that the load read from memory. Giri can then continue backtracking from the store to find instructions influencing the loaded value. Note that consulting the trace isn't required for finding the sources of values used by most LLVM instructions. Since most instructions operate on SSA scalar values, the input operands to those instructions can be determined from the explicit SSA graph in the LLVM IR representation of the program.

Finally, Giri instruments a program to record the execution of function calls and function returns. Having this information in the trace allows Giri to match up formal and actual arguments and caller/callee return values during the second phase for both direct and indirect function calls.

Giri calculates dynamic control dependences as follows: when Giri first adds instructions from a dynamic execution of a basic block to the backwards slice, it uses static control-dependence analysis [13] to determine which value forced the execution of that basic block. Giri will then find the most recent execution of the instruction generating that value and add it to the backwards slice.

By using the LLVM IR to reduce the trace size, our tool's traces are relatively small (43 to 144 MB in our experiments), making the use of more sophisticated optimizations unnecessary. To speed up the second phase, we perform one important optimization on the trace file. Giri performs a forward scan through the trace, recording the memory regions that have been modified by store instructions. If a load instruction is encountered that accesses a memory location not defined by a store (e.g., a non-modified global variable), it is given a special marking. This marking is used during backwards searches of the trace file to prevent tracing back to the beginning of the trace for loads without a defining store. Other optimizations are possible but are not implemented presently.

5. Filtering False Positives

In practice, dynamic backward slicing reduces the number of candidate root causes by a large amount (nearly 58%) but still leaves a fairly large number to be analyzed. We propose two new techniques to filter out false positives from the remaining set. One of these takes advantage of the dynamic tracing information, which allows us to filter invariants using the dynamic dependence graph (DDG), which is the union of the dynamic dataflow graph and the control dependence graph. The second one takes advantage of our input construction capability to identify candidates that appear to explain the symptom for multiple failing inputs that are only slightly different from the original one.

5.1 Dependent Chains of Failures

We observe that the result of an anomalous computation (i.e., one that violates an invariant) can cause dependent instructions to produce anomalous values. For example, if the variable `x` fails a range invariant because it is negative when it was expected to be non-negative, then the result of a subsequent computation using `x` (such as `2*x`) that was expected to be non-negative will be negative, too, potentially failing its range invariant. The real culprit, however, is the first failure; the second failure is just a false positive and not an independent buggy location.

We can filter out such false positives by identifying dependent chains of failing invariants using the DDG. The DDG is a graph $G(N, E)$, where the set of nodes N is the set of instructions executed in a given run of a program, and E is the set of edges $n_i \rightarrow n_j$ such that n_i is an instruction that generates a value that is an operand of n_j or n_j is control dependent on n_i. Graph G must be acyclic since it represents data flow and control dependence within a dynamic sequence of instructions. Conceptually, this graph can be constructed easily from the dynamic trace together with the program, since together they identify the complete set of CPU operations, loads and stores, and the dependencies between them.

We can use a standard depth-first-search traversal on this graph to look for cases where a failed invariant is dependent on a previous failed invariant. However, not all instructions have invariants computed for them. We therefore look for direct paths between two failed invariants without any intervening passing invariants, $n_1 \rightarrow n_2 \rightsquigarrow ... \rightsquigarrow n_{k-1} \rightarrow n_k$, $k \geq 2$, where (a) n_1 and n_k both have invariants that failed; and either (b) $k = 2$, or (b$'$) all instructions n_i, $2 \leq i \leq k-1$, do not have any passing invariants at all. In other words, there are no intervening invariants between n_1 and n_k at all. We then simply drop n_k from the set of candidate root causes under the argument that it "went bad" simply because n_1 went bad, as discussed above, i.e., it was not an independent bug location. Of course, n_1 may itself get dropped if it is, in turn, dependent on some previous instruction that had a failed invariant. Observe that the key case we are excluding with this rule is when there is a *passing* invariant between n_1 and n_k. Intuitively, we are "trusting" the invariants to distinguish correct values from erroneous ones. If an instruction has a passing invariant, we assume that it did not produce an erroneous value, and therefore a later dependent instruction that has a failed invariant may be an independent source of an error.

In practice, computing the DDG explicitly is unnecessary. Recall that the dynamic backward slicing algorithm, Giri, traverses the union of the dynamic dataflow graph and the control dependence graph of the execution. We piggyback directly on Giri to identify dependence chains. Giri essentially uses depth-first search (DFS) on the union of the two graphs to construct the backward slice. Every time Giri visits an instruction I, we check if I had an invariant and, if so, whether that invariant failed or passed. If I failed an invariant, we traverse I's parents in the depth-first-search tree (which corresponds to the chain of users) until we arrive at an ancestor instruction, A_I, that has a passing invariant. We mark all intervening instructions between A_I and I for later filtering. We also stop this ancestor traversal if we arrive at a failed invariant and mark it for later filtering. This ensures that we never need to process any instruction (i.e., mark it and recursively traverse its parents) more than once during the ancestor traversals, though we may arrive at it as many times as there are incoming operands (i.e., incoming edges in the DDG). This ensures that the total time for this dependence filtering pass is linear in the number of edges in the DDG.

At the end of this algorithm, we filter out all remaining candidate instructions that were marked for filtering. In practice, this pass is also very effective at eliminating false positives. It is unfortunately possible, however, that this pass may occasionally eliminate a true root cause itself. In particular, it is possible that I_2 depends on I_1, both have failed invariants, but I_1 was actually a false positive while I_2 was a true root cause. However, we expect this behavior to be rare with sufficient training.

5.2 Multiple Failure-inducing Inputs

Although we found that the number of remaining candidate root causes after dependence filtering tends to be fairly low (e.g., in the range of 7-18, reduced from hundreds originally), we can do better for some bugs without new mechanisms. We observe that, using essentially the same methodology that we used for nearby good inputs, we can construct additional failure-inducing inputs that are close to the original one *and produce the same failure symptom as the original one*. For each such input, since the failure symptom is the same *and* the inputs are very similar to the original one by construction, the true root cause of the failure is very likely to be the same.

This insight leads to the following simple strategy. Construct a set of additional failure-inducing inputs close to the original one that produce the same failure symptom. Repeat the previous analysis for each such new input: (1) identify all failing invariants when executing with that input (using the invariants previously extracted using the set of good inputs); (2) filter out invariants not on the dynamic backward slice of the failure symptom; and (3) filter out invariants using the dependence filtering step above. This process gives a set of candidate invariants for this failing input. Compute the intersection of these sets from all the failing inputs. That intersection yields the set of invariants that are candidate root causes for all failing inputs and is the final output of our analysis.

6. Generating Programmer Results

Each candidate invariant in the final set represents an LLVM instruction that may indicate the location of a root cause. Recall, however, that we only track invariants on load, store and return instructions. This means that we must identify the entire subexpression rooted at a load, store or return for the programmer.

We identify the operations involved in the subexpression using a simple traversal of the Dynamic Dependence Graph. We could place invariants on these subexpression values to remove parts of the subexpression that are not the root cause, but we do not have this implemented at present. Once all LLVM instructions within the subexpression are identified, our analysis maps them back to source statements using debug information.

7. Experimental Methodology

7.1 Applications

In our experiments, we used three widely-used servers: the Squid HTTP proxy server, the MySQL database server, and the Apache HTTP web server. MySQL, which has about a million lines of code, is by far the largest application used to evaluate automatic diagnosis algorithms in the literature to date, except in the Triage work [42]. All these servers communicate with clients using well-defined protocols with well-defined grammars amenable to our input construction algorithm.

7.2 Software Bugs and Failure Detectors

We selected 8 real software bugs for these three server programs. We were limited in two ways. First, we needed bugs that we could reproduce using information in the bug reports for versions of the software that we could compile with LLVM on a current Linux system. Second, our current implementation cannot handle concurrency bugs as we need a deterministic detector. Our implementation also cannot handle missing code bugs (usually uninitialized variable bugs) because the root cause may not be effectively identified by range invariants. We expect they will need control flow invariants, which are outside the scope of this work. Hence, out of 12 such bugs, we omitted four "missing code" bugs; we plan to investigate control flow invariants for diagnosing such bugs in future.

Table 1 describes in detail the eight bugs we used. The first column provides a name for each bug for identification, using which we refer to the bugs later. The second column shows the name of the application and version which contains the bug. The third column provides the number of lines of code (in units of 1000 lines). The next column shows the failure symptom of the bug: four bugs were memory safety errors; the rest were incorrect output bugs. The last column provides a brief description of each bug.

The selected bugs have different root causes. For example, some bugs occur due to integer overflow or loss of precision. One buffer overflow bug occurred due to the use of an unsigned intermediate value for some computation. buffer overflow bug occurred when a buffer length was incorrectly computed for some specific inputs. In another bug, using an incorrect MySQL object type for a MySQL item causes a segmentation fault.

The difficulty of fault localization will depend upon many properties of the bug. Table 2 lists some important properties of these 8 bugs. The second column shows the unique, static number of source lines of code that are executed for the failure-inducing input. We first use the program trace to determine the LLVM instructions

Table 2. Bug Characteristics which may Impact Difficulty of Diagnosis.

#Bug	Static #LOC executed in failed run	Distance (Dyn #LLVM inst)	Distance (Static #LOC)	Distance (Static #Functions)
Bug-1	6927	12	6	2
Bug-2	9822	18	7	3
Bug-3	9982	86	27	10
Bug-4	11308	4	4	2
Bug-5	7874	124	41	17
Bug-6	7835	114	36	17
Bug-7	9835	429	16	5
Bug-8	6217	32780	6	2

that are executed during the failing run, and then use LLVM debug metadata information to map them to their corresponding program source lines. For some LLVM instructions, debug metadata may be missing due to optimizations. We count one source line for each such instruction. All the bugs require execution of thousands of lines of source code before resulting in the failure symptom. The number of executed source lines is an indicator of the difficulty of fault localization. Note that the complexity of fault localization for an automated tool will also depend upon the size or lines of code of the complete application. The next three columns specify another important property which can impact the difficulty of diagnosis process – the distance from the symptom to the root cause. The third column gives the distance from symptom to root cause along the dynamic slice in terms of number of dynamic LLVM instructions. LLVM instructions are simple instructions, each of which translates to approximately 2-3 x86 instructions on average [6]. The next two columns show the static number of source lines and functions corresponding to these dynamic LLVM instructions. The root causes for incorrect output bugs are sometimes far from the symptom. More importantly, the intervening statements along the slice span several different functions, increasing the difficulty of the fault localization process. Note that these numbers do not include many other intervening statements and functions which are outside the path from root cause to symptom. Our approach, on the other hand, is agnostic to the distance of root cause from symptom, unlike other previous work like Triage [42].

We do not use automated detectors for our experiments. For each memory safety error, we manually inserted program assertions at the program point where a tool like SAFECode would have detected a memory safety error. For each incorrect output error, we manually inserted an assertion at the program point just before the server starts processing the incorrect output to send it to the client. In both cases, the assertions serve both as detectors and as starting points for dynamic backwards slicing.

Although we use manually inserted detectors, we expect that insertion of automatic detectors will be feasible in practice. One of the most important use cases for our tool is automated testing (e.g., such as nightly testing, which is widely used), which already makes use of automatic failure detectors, including "known good outputs" that identify wrong-output errors. For memory safety errors or integer numerical errors, we can also use a tool like SAFECode [14] or IOC [15], which precisely detect the symptoms of the failure. For incorrect output errors outside automated testing, comparison with the output of a different version of the application or other equivalent software may also work as a detector. Alternatively, we can fall back on programmer-inserted assertions for detection.

7.3 Good and Bad Input Construction

Our proposed diagnosis approach begins with these steps:

1. Compute a small failure-inducing input from the original failure-inducing input that fails with the same symptom.

Table 1. Software Bugs Used In The Experiments.

Bug#	Application	LOC	Symptom	Bug Description
Bug-1	Squid 2.3	70K	buffer overflow	Incorrect computation of buffer length leads to buffer overflow
Bug-2	MySQL 5.1.30	1019K	buffer overflow	Use of unsigned variable causes integer overflow which leads to buffer overflow
Bug-3	MySQL 5.1.30	1019K	Incorrect output	Wrong algorithm to convert microsecond field to integer results in incorrect value
Bug-4	MySQL 5.1.23a	1054K	buffer overflow	Use of a negative number along with an aggregate function results in seg fault
Bug-5	MySQL 5.0.18	949K	Incorrect output	Loss of precision in a sequence of computation results in wrong value
Bug-6	MySQL 5.0.18	949K	Incorrect output	Overflow during decimal multiplication results in garbage output
Bug-7	MySQL 5.0.15	937K	Incorrect output	Loss of data when inserting big values in a table
Bug-8	Apache 2.2	225K	buffer overflow	For particular value of output size, buffer overflow occurs

2. Construct a small set of good inputs that are close to the small failure-inducing input.
3. Construct a small set of additional failure-inducing inputs that fail with exactly the same symptom as the original failing input.

For step 1, we already had fairly small failure-inducing inputs available from the bug reports for six of the eight bugs. For the remaining two bugs, we slightly reduced the inputs by hand by applying an algorithm similar to *ddmin* [45], but this reduction is not critical to our results. For steps 2 and 3, we constructed 8 good and 8 bad inputs for the server bugs using the input construction algorithms described in Section 3.2. For the Apache bug (bug-8), our current approach could not generate additional bad inputs, as only very few specific input values could trigger the failure. To determine if an input was a good input, we tested the input on the buggy version of the application and on a version of the application with the bug fixed.[3] If the fixed application returned errors on the input, we assumed the input was grammatically incorrect and threw it out. All other inputs either did not trigger the bug and produced identical output on both the buggy and fixed versions of the application (i.e., were deemed as good inputs) or produced correct output in the fixed version but triggered the bug in the buggy version (i.e., were deemed as bad inputs). This approach allowed us to categorize inputs for incorrect output bugs sans an automated error detector.

It should be noted that our approach does not require "minimal" failure-inducing input. In many cases, developers already have reduced inputs which should suffice for our approach. The inputs from the bug reports we used in experiments were small but not necessarily minimal. An earlier empirical study [37] of over 260 randomly selected bugs in six different server applications (including some stateful servers) found that most bugs could be reproduced with 3 or fewer inputs. If the inputs are very large, tools like ddmin [45], delta debugging [12, 44], and C-Reduce [36] do a good job of minimizing the inputs for many classes of applications. Hence, it will not be hard to get small enough inputs for most bugs for successful diagnosis.

8. Experimental results

In this section, we evaluate the effectiveness of our approach. In particular, we investigate (a) whether our approach can find the true root causes of bugs; and (b) how many false positives it generates. For the purpose of this evaluation, we use the following definition of root cause: for each bug, we use the correct patch in the bug reports to identify the minimal statements which should be changed or deleted to remove the failure symptom. (The only cases where new lines of code are added are to replace some existing lines, since we do not include missing-code bugs in our experiments.) We say all these statements are part of the root cause, and we say our approach is successful if it can identify all these statements.

[3] This was purely for experimental evaluation; in practice, any other suitable approach like automated memory-error detectors, programmer-inserted assertions or comparison with known-output can be used to determine whether the failure symptom is avoided or not.

We used the LLVM-2.6 compiler [29] infrastructure to compile the programs, to extract invariants and compute dynamic backward slices. We conducted the experiments on Linux 2.6 on a machine with a dual-core Intel ® Core ™ 2 Duo CPU running at 3 GHz with 4 MB cache and 8 GB of RAM.

The time for the complete diagnosis depends on the trace size and program size. For these bugs, total diagnosis time varied from around 8 minutes up to 4 hours depending upon the application. For two bugs, we believe it took an excessively long time because our slicing implementation is not optimized. We expect to see many fold improvements in the execution time after a few optimizations.

The detailed experimental results using automated input generation are presented in Table 3. The first column in the table lists the bug numbers from Table 1. The second column shows the total number of invariants that are generated and inserted by our system. The third column shows the total number of failed invariants by the original failure-triggering input. For memory bugs where failure occurs soon after root cause, there may not be any invariant present between the root cause and failure symptom to catch divergent behavior. Failure location can give a strong clue about root cause for such cases. Hence, for such kinds of bugs, we also count the program statement at the failure location as a candidate root cause and include it in this column. The fourth column shows the number of failed invariants that are on the dynamic backward slice. The fifth column shows the number after the dependence filtering step, and the sixth column shows the number after the filtering using multiple failure-inducing inputs. The final column shows the maximum number of source lines corresponding to all the expression trees which the developer needs to analyze to fix the bug depending upon which latest step contains the root cause. The last column shows whether the candidates after the final multiple inputs filtering step includes the root cause or not.

Note that a single candidate sub-expression at the LLVM level may span multiple source code lines. Therefore, the number of source statements is, in general, larger than the number of final candidate locations. We believe, however, that the difficulty of the programmer's final task is best measured by the number of candidate locations they have to understand, and not simply the total number of source code lines. In particular, the number of source lines highly depends on the idioms the programmer uses rather than the fundamental nature of the candidate computations: a relatively simple sub-expression may span two or more lines of code or a complex computation may be expressed in a single compact statement. For this reason, we measure the effectiveness of our analysis primarily by the number of candidate locations and secondarily by the number of source statements they represent.

8.1 Number of Candidate Root Causes

From the results, it is clear that dynamic backward slicing is effective in reducing the number of candidate root causes, consistently removing nearly 80% of the false positives. In three cases, it reduces them by factors of 6 (Bug 6) and 10 (Bug 1 and 8).

The dependence filtering step is also very effective in reducing the remaining false positives. On average, it removes nearly 58%

Table 3. Number of Reported Root Causes

Bug#	#Invs	Failed Invs	Slice	Dependence filter	Multiple faulty inputs	Src-expr-tree	Root Cause in final step?
Bug-1	3358	357	30	9	9	49	Yes
Bug-2	5917	95	36	16	12	48	No
Bug-3	5942	93	27	9	6	64	Yes
Bug-4	6847	156	44	14	8	28	Yes
Bug-5	4566	130	34	18	17	89	Yes
Bug-6	4652	83	13	7	5	26	Yes
Bug-7	5836	153	35	17	11	152	Yes
Bug-8	2295	120	12	6	6	171	Yes

of the false positive candidate root causes, in one case (Bug 1) removing nearly 70% of them.

The last filtering step, using multiple faulty inputs, has less consistent benefits. In some cases (Bugs 1 and 5), it has only a small effect or no effect at all. In other cases, the benefit is quite substantial (Bugs 2, 3, 4 and 7), especially considering that it comes after a series of other highly effective filters.

8.2 Efficacy at Finding True Root Causes

As Table 3 shows, when using all of the false positive filters, our tool included the root cause in the final list of candidates for 7 of the 8 bugs. Our approach could not narrow down the root cause for Bug 2 among the program expressions in the final filtering step. However, the root cause is still included in the candidate set after the dependence filtering step. Hence, developers need to analyze 16 candidate program expressions instead of 12 expressions to find the exact root cause. We found that diagnosis for this bug is very sensitive to input. When we tried a different set of training inputs, the final step was able to identify the root cause, essentially because the training runs produced a more precise set of invariants.

There are at least two strategies to deal with such "false negative" cases. The programmer can investigate the final suggestions first and, when all are eliminated, ask the tool for other suggestions. The tool could present the penultimate set of 16 locations (4 additional locations compared to initial 12 locations) in this case, and the user would investigate four more locations to find the bug. Second, the tool could be made more effective with a better input generation algorithm which can identify the root cause for all cases.

Note that our approach is more likely to identify root causes of bugs, including Bug 2, than invariants extracted using general inputs. For example, using our input construction algorithm, we are able to find the root cause for Bug 1, where a large input field (a *username*) with many special characters (as in a ftp request like $ftp : //usernam\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash\backslash:$ $pass@ftp.cs.wisc.edu$) causes a segmentation fault. Each special character needs three times the buffer size compared to normal characters; however, the code incorrectly allocates less buffer size leading to a buffer overflow. If we use a general set of training inputs, the invariant ranges on the buffer length would be wider and bug-independent, making them less effective. More specifically, if any HTTP training input contained a larger *username* field than the failure-inducing input, the analysis would miss the root cause. However, our invariants with a better and focused training set could find the root cause through invariants on buffer length.

It is also important to note that our approach missed the most ideal invariant failure which could have accurately pinpointed the root cause in bug-7. This happens because the value on which invariant fails is not used in any computation, hence it gets filtered out. However fortunately, a different, but related invariant fails which was able to identify the actual root cause.

For bugs 7 and 8, the number of source lines are comparatively larger as a few expression trees inside large functions accounted for a much larger number of source lines. However, this approach

Table 4. Tarantula/Ochiai Results

Bug	Tarantula	Ochiai	Our Approach
Bug-1	[6 - 5266]	[2 - 5255]	49
Bug-2	[34 - 34]	[34 - 34]	48
Bug-3	[55 - 9409]	[55 - 9409]	64
Bug-4	[776 - 9847]	[694 - 9762]	28
Bug-5	[1 - 19]	[1 - 16]	89
Bug-6	[5680 - 6878]	[5678 - 6876]	26
Bug-7	[8 - 6060]	[8 - 6060]	152
Bug-8	[28 - 5372]	[1 - 5357]	171

still results in a significant reduction considering the fact that over 9000 and 6000 static source lines are executed by these two bugs respectively. For example, the six LLVM-level expressions of bug-8 mapped to 2, 11, 15, 35, 92, and 115 source-level statements respectively (note that some of the expressions overlap). The root-cause was in the third expression spanning only 15 source lines. The latter two, very high numbers, greatly skewed the average, and both are false positives. For this bug, we could not apply the last filtering step as our current approach was not able to create additional bad inputs. Similarly, three large false-positive expressions increase the final number of source lines for bug-7 significantly though the expression containing the true root-cause consists of 16 source lines. Further elimination of false positives would greatly bring down the average number of source lines in these bugs as well. Moreover, we would like to add a second pass of invariants and filtering that evaluates individual *intermediate values within expressions* to narrow down the root causes internal to these expressions. This should greatly bring down the final number of source statements reported to the programmer.

8.3 Comparison with Tarantula and Ochiai

The Tarantula [26, 27] and Ochiai [5] statistical approaches, while simplistic in nature, are powerful methods that allow for reproduction in a reasonable amount of time and serve as a good baseline for software bug diagnosis. These two approaches work by performing an analysis of executed program statements for a given set of inputs. Specifically, they record for each input whether or not the faulty behavior was induced and the statements which were executed. This results in a vector for each program statement whose elements are 1 if the statement executed for a given input and 0 if it did not. These vectors are compared against a success/failure vector whose elements are 1 if the program exhibited the failure and 0 if it did not. A similarity coefficient is computed using each approach's namesake formula, where each a_{ij} is a counter for the number of elements in the source line vector with value i whose corresponding element in the success/failure vector has value j.

$$tarantula = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}}$$

$$ochiai = \frac{a_{11}}{\sqrt{(a_{11}+a_{01}) * (a_{11}+a_{10})}}$$

Program statements are then ranked in decreasing order by this similarity score. The basic principle serves to identify which program statements are highly correlated with program failures. We include Ochiai in our comparison because this similarity metric has been found to give better empirical results [5]. We have implemented these approaches using the LLVM compiler system, leveraging the trace files generated by Giri.

Table 4 lists, for each of our bugs, the range of source lines a programmer would have to inspect to find the root cause. We computed this by determining the similarity metric for each source line, ranking them, and then counting all statements that had a higher ranking than the root cause. In most cases, the Tarantula or Ochiai formulas will calculate many source lines as having the same similarity; this is reflected by the ranges offered, where the best case occurs if the root cause is listed at the top of the list of ties, and the worst case when it is listed at the bottom.

We see that the statistical correlation method of bug diagnosis' effectiveness varies dramatically depending on the type of bugs analyzed. In particular, when analyzing bugs that involve program control-flow divergence (such as Bug 5), we see that the Tarantula and Ochiai approaches perform notably well, but in cases where the same root cause is executed under both successful and failure-inducing inputs (such as Bug 6), we see results that are much less impressive, as one would expect. This highlights an important shortcoming of the correlation-based approaches: they fail to accurately account for cases where there is an absence of control flow divergence that leads to failure-inducing behavior. In our sampling of all bugs, the statistical methods only performed well on 2 out of 8 bugs—our approach provides much more consistent behavior.

8.4 Summary and Limitations

In summary, the analysis tool is able to identify only 5 to 17 candidate root cause expressions for all 8 bugs in three programs, one of which is millions of lines long. These results indicate that bug-specific likely invariants and dynamic backwards slicing, combined with our novel filtering techniques, can be extremely effective in helping programmers diagnose the root causes of failures.

Our current implementation also has some limitations. First, it cannot effectively handle missing code bugs like missing initialization and concurrency bugs as explained earlier. Second, various stages of the approach depend on robust input generation, as seen in one bug that was diagnosed with one set of inputs but, with another set, the root cause was dropped by the "multiple faulty inputs" filter. Also, for Bug 2, in an early version of our system that had *insufficient* training, dependence filtering removed the root cause because it was directly dependent on a false positive. This behavior has not happened since we improved the automated input generation procedure. Similarly, the "multiple faulty inputs" filter in an early version of our system removed the root cause for Bug 3, but this did not occur with improved input generation. Finally, we do not yet generate invariants for function arguments, which can make some of our expression trees unnecessarily large.

We can also combine our approach with other static and dynamic analysis approaches e.g., Tarantula. Moreover, our analysis computes but does not report a lot of valuable information (like the invariant failures and values) that could further assist diagnosis bugs. Such post-processing is outside the scope of this work.

9. Related Work

Delta debugging [12, 44] compares a successful and a failing run for fault localization but does not scale to larger programs due to inaccurate mapping of program states. Recent work on program execution indexing and memory indexing [41, 43] has greatly improved delta debugging's diagnostic accuracy. However, these techniques compute causal paths instead of root causes, and previous work [39, 40] shows 45% of the computed causal paths miss the root cause. Previous work [39, 40] also states that if a different input is used for a failing run, it can give inaccurate causal paths. Also, when copying state from a correct run to a faulty run, it may be necessary to copy more than the faulty state, thus resulting in false positives. It is also not clear how this approach will scale for larger programs and larger program traces.

DIDUCE [24] was the first work to use invariants for bug diagnosis; it used a variant of range invariants to find root causes. Pytlik et.al. [35] used invariants for fault localization of Siemens benchmark suite without much success possibly due to the fact that they used general test inputs to generate invariants. Statistical techniques like Tarantula and others [5, 26, 27] use differences in control flow between successful and failing runs and rank the statement based on a suspiciousness criteria. Liblit et. al. [30] developed more efficient methods to use statistical techniques with much lower overhead. However, these stand-alone techniques can result in too many false positives. Mariani et. al. [31] filter out invariants which are also violated in successful executions as false-positives, which may be less effective with good training.

Dynamic slicing has also been used for bug diagnosis; failure inducing chop [23] used both forward and backward slicing. Triage [42] combines differences in control flow between successful and failing runs with dynamic backward slicing. We improve upon these techniques by combining dynamic slicing with invariants similar to the work by Dimitrov et. al. [16]. We improve upon Dimitrov et. al. [16] in that our techniques to reduce false positives are not ad hoc, our system does not require special hardware support, and our system utilizes control dependence which, while it can increase false positives, is needed for diagnosing certain bugs [47].

BugAssist's [28] approach of using MAX-SAT solvers for fault localization is more systematic than ours. Execution perturbations [25, 46] have been used to find root causes for small programs. We do not believe that these techniques alone will scale to larger programs. Our approach is orthogonal to these approaches and can be combined with them to improve diagnostic accuracy.

Fuzzing techniques [9, 19] have previously been used to randomly modify inputs to create new inputs. Recent advances in symbolic execution and dynamic test generation techniques [21] have given rise to a new form of testing called whitebox fuzzing technique [22] to create inputs for discovering new software bugs. Artzi et. al. [7, 8] also developed techniques to automatically generate test cases using a form of concolic testing which they used for fault localization of PHP applications using known statistical techniques. Our emphasis is the opposite: our focus is more on effective fault localization techniques than on input generation. We generate inputs by modifying input according to an input specification. Our strategy is less general, but more scalable for larger programs. We also showed that using similar inputs in an invariants-based framework can, in fact, increase diagnosis accuracy by localizing faults which will be missed by using general test inputs. However, our tool could use these techniques to generate inputs for applications in which our input generation algorithms do not yield good results.

Finally, our work, to the best of our knowledge, performs diagnosis on much larger applications than any previous automated diagnosis technique except Triage [42].

10. Future Work and Conclusions

We have proposed a new automated bug diagnosis approach that combines invariant-based diagnosis, delta debugging, and dynamic backward slicing to compute an accurate set of possible root causes. Our system automatically constructs good inputs which are similar to the failure-inducing input to develop better invariants that are more likely to find root causes. We also proposed two new heuristics that play an important role in reducing false positives. The overall procedure is able to narrow down the root causes of bugs in large, real-world applications to only 5–17 program expressions, even in programs with hundreds of thousands of lines of code.

In the future, we plan to make our automated input construction algorithms more robust and evaluate it with more applications and bugs. We also plan to investigate new and other existing program analysis techniques to reduce false positives further, and new invariants which can capture a wider variety of bugs like concurrency bugs and missing code bugs.

Acknowledgments

We gratefully acknowledge the support by NSF contracts CCF 08-11693 and CNS 12-19080 for this work.

References

[1] Website. http://findbugs.sourceforge.net/.

[2] Website. http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-fortify-static-code-analyzer/.

[3] Website. http://www.coverity.com/products/coverity-save.html.

[4] NIST: National Institute of Standards and Technology. Software errors cost U.S. economy $59.5 billion annually: NIST assesses technical needs of industry to improve software-testing. June 2002.

[5] R. Abreu, P. Zoeteweij, and A. J. van Gemund. An evaluation of similarity coefficients for software fault localization. In *12th Pacific Rim International Symposium on Dependable Computing(PRDC)*, 2006.

[6] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *Proc. ACM/IEEE Int'l Symp. on Microarch. (MICRO)*, page 205, Washington, DC, USA, 2003. IEEE Computer Society.

[7] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10.

[8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010.

[9] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229 –245, 1983.

[10] M. Bond. *Diagnosing and Tolerating Bugs in Deployed Systems*. PhD thesis, University of Texas at Austin, 2008.

[11] R. N. Charette. Why Software Fails. *Spectrum, IEEE*, 42(9):42–49, September 2005.

[12] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, pages 13(4):451–490, October 1991.

[14] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[15] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding Integer Overflow in C/C++. In *International Conference on Software Engineering*, June 2012.

[16] M. Dimitrov and H. Zhou. Anomaly-based bug prediction, isolation, and validation: an automated approach for software debugging. In *ASPLOS*, 2009.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, 2000.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 2001.

[19] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium*, 2000.

[20] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. C. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *SOSP*, 2009.

[21] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, 2005.

[22] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[23] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, 2005.

[24] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, 2002.

[25] D. Jeffrey, N. Gupta, and R. Gupta. Effective and efficient localization of multiple faults using value replacement. In *ICSM*, 2009.

[26] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, 2002.

[27] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[28] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, 2011.

[29] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization*, 2004.

[30] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[31] L. Mariani, F. Pastore, and M. Pezze. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37:486–508, 2011.

[32] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *International Symposium on Empirical Software Engineering and Measurement*, 2009.

[33] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44(6):245–258, 2009.

[34] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2), 2003.

[35] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. *In Proceedings of the Workshop on Automated and Algorithmic Debugging*, 2003.

[36] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellision, and X. Yang. Test-case reduction for c compiler bugs. In *Submission*, 2012.

[37] S. K. Sahoo, J. Criswell, and V. Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *ICSE*, 2010.

[38] D. Scott. Making smart investments to reduce unplanned downtime. *Tactical Guidelines TG-07-4033, Gartner Group*, March 1999.

[39] W. N. Sumner and X. Zhang. Automatic failure inducing chain computation through aligned execution comparison. In *Technical Report 08-023, Purdue University*, 2008.

[40] W. N. Sumner and X. Zhang. Algorithms for automatically computing the causal paths of failures. In *FASE*, 2009.

[41] W. N. Sumner and X. Zhang. Memory indexing: canonicalizing addresses across executions. In *FSE*, 2010.

[42] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, 2007.

[43] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI*, 2008.

[44] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02: FSE*, 2002.

[45] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2002.

[46] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, 2006.

[47] X. Zhang, N. Gupta, and R. Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 2007.

[48] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.