Scalable Statistical Bug Isolation

Ben Liblit
Computer Sciences Department
University of Wisconsin-Madison
<liblit@cs.wisc.edu>

Mayur Naik
Computer Science Department
Stanford University
<mhn@cs.stanford.edu>

Alice X. Zheng
Department of Electrical Engineering
and Computer Science
University of California, Berkeley
<alicez@cs.berkeley.edu>

Alex Aiken
Computer Science Department
Stanford University
<aiken@cs.stanford.edu>

Michael I. Jordan
Department of Electrical Engineering
and Computer Science
Department of Statistics
University of California, Berkeley
<jordan@cs.berkeley.edu>

Abstract

We present a statistical debugging algorithm that isolates bugs in programs containing multiple undiagnosed bugs. Earlier statistical algorithms that focus solely on identifying predictors that correlate with program failure perform poorly when there are multiple bugs. Our new technique separates the effects of different bugs and identifies predictors that are associated with individual bugs. These predictors reveal both the circumstances under which bugs occur as well as the frequencies of failure modes, making it easier to prioritize debugging efforts. Our algorithm is validated using several case studies, including examples in which the algorithm identified previously unknown, significant crashing bugs in widely used systems.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—statistical methods; D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, distributed debugging, monitors, tracing; I.5.2 [*Pattern Recognition*]: Design Methodology—feature evaluation and selection

General Terms Experimentation, Reliability

Keywords bug isolation, random sampling, invariants, feature selection, statistical debugging

[*] This research was supported in part by NASA Grant No. NAG2-1210; NSF Grant Nos. EIA-9802069, CCR-0085949, ACI-9619020, and ENG-0412995; DOE Prime Contract No. W-7405-ENG-48 through Memorandum Agreement No. B504962 with LLNL; DARPA ARO-MURI ACCLIMATE DAAD-19-02-1-0383; and a grant from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006...$5.00.

1. Introduction

This paper is about *statistical debugging*, a dynamic analysis for identifying the causes of software failures (i.e., bugs). Instrumented programs monitor their own behavior and produce feedback reports. The instrumentation examines program behavior during execution by sampling, so complete information is never available about any single run. However, monitoring is also lightweight and therefore practical to deploy in a production testing environment or to large user communities, making it possible to gather information about many runs. The collected data can then be analyzed for interesting trends across all of the monitored executions.

In our approach, instrumentation consists of *predicates* tested at particular program points; we defer discussing which predicates are chosen for instrumentation to Section 2. A given program point may have many predicates that are sampled independently during program execution when that program point is reached (i.e., each predicate associated with a program point may or may not be tested each time the program point is reached). A feedback report R consists of one bit indicating whether a run of the program succeeded or failed, as well as a bit vector with one bit for each predicate P. If P is observed to be true at least once during run R then $R(P) = 1$, otherwise $R(P) = 0$.

Let B denote a bug (i.e., something that causes incorrect behavior in a program). We use $\mathcal{B}$ to denote a *bug profile*, i.e., a set of failing runs (feedback reports) that share B as the cause of failure. The union of all bug profiles is exactly the set of failing runs, but note that $\mathcal{B}_i \cap \mathcal{B}_j \neq \emptyset$ in general; more than one bug can occur in some runs.

A predicate P is a *bug predictor* (or simply a *predictor*) of bug B if whenever $R(P) = 1$ then it is statistically likely that $R \in \mathcal{B}$ (see Section 3.1). *Statistical debugging* selects a small subset $\mathcal{S}$ of the set of all instrumented predicates $\mathcal{P}$ such that $\mathcal{S}$ has predictors of all bugs. We also rank the predictors in $\mathcal{S}$ from the most to least important. The set $\mathcal{S}$ and associated metrics (see Section 4) are then available to engineers to help in finding and fixing the most serious bugs.

In previous work, we focused on techniques for lightweight instrumentation and sampling of program executions, but we also studied two preliminary algorithms for statistical debugging and presented experimental results on medium-size applications with a single bug [10, 16]. The most general technique we studied is *regularized logistic regression*, a standard statistical procedure that tries to select a set of predicates that best predict the outcome of every run. As we worked to apply these methods to much larger programs under realistic conditions, we discovered a number of serious scalability problems:

- For large applications the set $\mathcal{P}$ numbers in the hundreds of thousands of predicates, many of which are, or are very nearly, logically redundant. In our experience, this redundancy in $\mathcal{P}$ causes regularized logistic regression to choose highly redundant lists of predictors $\mathcal{S}$. Redundancy is already evident in prior work [10] but becomes a much more serious problem for larger programs.
- A separate difficulty is the prevalence of predicates predicting multiple bugs. For example, for many Unix programs a bug is more likely to be encountered when many command line flags are given, because the more options that are given non-default settings the more likely unusual code paths are to be exercised. Thus, predicates implying a long command line may rank near the top, even though such predicates are useless for isolating the cause of individual bugs.
- Finally, different bugs occur at rates that differ by orders of magnitude. In reality, we do not know which failure is caused by which bug, so we are forced to lump all the bugs together and try to learn a binary classifier. Thus, predictors for all but the most common bugs have relatively little influence over the global optimum and tend to be ranked very low or not included in $\mathcal{S}$ at all.

These problems with regularized logistic regression persist in many variations we have investigated, but analysis of this body of experimental work yielded some key technical insights. In addition to the bug predictors we wish to find among the instrumented predicates, there are several other kinds of predicates. First, nearly all predicates (often 98% or 99%) are not predictive of anything. These *non-predictors* are best identified and discarded as quickly as possible. Among the remaining predicates that can predict failure in some way, there are some bug predictors. There are also *super-bug predictors*: predicates that, as described above, predict failures due to a variety of bugs. And there are *sub-bug predictors*: predicates that characterize a subset of the instances of a specific bug; these are often special cases of more general problems. We give the concepts of super- and sub-bug predictors more precise technical treatment in Section 3.3.

The difficulty in identifying the best bug predictors lies in not being misled by the sub- or super-bug predictors and not being overwhelmed by the sheer number of predicates to sift through. This paper makes a number of contributions on these problems:

- We present a new algorithm for isolating multiple bugs in complex applications (Section 3) that offers significant improvements over previous work. It scales much more gracefully in all the dimensions discussed above and for each selected predicate P it naturally yields information that shows both how important (in number of explained program failures) and how accurate a predictor P is.
- We validate the algorithm by a variety of experiments. We show improved results for previously reported experiments [10]. In a controlled experiment we show that the algorithm is able to find a number of known bugs in a complex application. Lastly, we use the algorithm to discover previously unknown serious crashing bugs in two large and widely used open source applications.
- We show that relatively few runs are sufficient to isolate all of the bugs described in this paper, demonstrating that our approach is feasible for in-house automatic testing as well as for deployment to end users (see Section 4.3).
- We report on the effectiveness of the current industry practice of collecting stack traces from failing runs. We find that across all of our experiments, in about half the cases the stack is useful in isolating the cause of a bug; in the other half the stack contains essentially no information about the bug's cause.
- Finally, we show that, in principle, it is possible for our approach to help isolate any kind of failure, not just program crashes. All that is required is a way to label each run as either "successful" or "unsuccessful."

With respect to this last point, perhaps the greatest strength of our system is its ability to automatically identify the cause of many different kinds of bugs, including new classes of bugs that we did not anticipate in building the tool. By relying only on the distinction between good and bad executions, our analysis does not require a specification of the program properties to be analyzed. Thus, statistical debugging provides a complementary approach to static analyses, which generally do require specification of the properties to check. Statistical debugging can identify bugs beyond the reach of current static analysis techniques and even new classes of bugs that may be amenable to static analysis if anyone thought to check for them. One of the bugs we found, in the RHYTHMBOX open source music player, provides a good illustration of the potential for positive interaction with static analysis. A strong predictor of failure detected by our algorithm revealed a previously unrecognized unsafe usage pattern of a library's API. A simple syntactic static analysis subsequently showed more than one hundred instances of the same unsafe pattern throughout RHYTHMBOX.

The rest of the paper is organized as follows. After providing background in Section 2, we discuss our algorithm in Section 3. The experimental results are presented in Section 4, including the advantages over our previous approach based on regularized logistic regression. Section 5 considers variations and extensions of the basic statistical debugging algorithm. We discuss related work in Section 6 and offer our conclusions in Section 7.

2. Background

This section describes ideas and terminology needed to present our algorithm. The ideal program monitoring system would gather complete execution traces and provide them to an engineer (or, more likely, a tool) to mine for the causes of bugs. However, complete tracing of program behavior is simply impractical; no end user or tester would accept the required performance overhead or network bandwidth.

Instead, we use a combination of sparse random sampling, which controls performance overhead, and client-side summarization of the data, which limits storage and transmission costs. We briefly discuss both aspects.

Random sampling is added to a program via a source-to-source transformation. Our sampling transformation is general: any collection of statements within (or added to) a program may be designated as an *instrumentation site* and thereby sampled instead of run unconditionally. That is, each time instrumentation code is reached, a coin flip decides whether the instrumentation is executed or not. Coin flipping is simulated in a statistically fair manner equivalent to a Bernoulli process: each potential sample is taken or skipped randomly and independently as the program runs. We have found that a sampling rate of 1/100 in most applications[1] keeps the performance overhead of instrumentation low, often unmeasurable.

[1] Some compute-bound kernels are an exception; we currently sometimes resort to simply excluding the most performance critical code from instrumentation.

Orthogonal to the sampling transformation is the decision about what instrumentation to introduce and how to concisely summarize the resulting data. Useful instrumentation captures behaviors likely to be of interest when hunting for bugs. At present our system offers the following instrumentation schemes for C programs:

branches: At each conditional we track two predicates indicating whether the true or false branches were ever taken. This applies to `if` statements as well as implicit conditionals such as loop tests and short-circuiting logical operators.

returns: In C, the sign of a function's return value is often used to signal success or failure. At each scalar-returning function call site, we track six predicates: whether the returned value is ever < 0, ≤ 0, > 0, ≥ 0, $= 0$, or $\neq 0$.

scalar-pairs: Many bugs concern boundary issues in the relationship between a variable and another variable or constant. At each scalar assignment `x = ...`, identify each same-typed in-scope variable $\mathtt{y}_i$ and each constant expression $\mathtt{c}_j$. For each $\mathtt{y}_i$ and each $\mathtt{c}_j$, we track six relationships to the new value of x: $<, \leq, >, \geq, =, \neq$. Each $(\mathtt{x}, \mathtt{y}_i)$ or $(\mathtt{x}, \mathtt{c}_j)$ pair is treated as a distinct instrumentation site—in general, a single assignment statement is associated with multiple distinct instrumentation sites.

All predicates at an instrumentation site are sampled jointly. To be more precise, an *observation* is a single dynamic check of all predicates at a single instrumentation site. We write "*P* observed" when the site containing *P* has been sampled, regardless of whether *P* was actually true or false. We write "*P* observed to be true" or merely "*P*" when the site containing *P* has been sampled and *P* was actually found to be true. For example, sampling a single negative return value means that all six returns predicates (< 0, ≤ 0, > 0, ≥ 0, $= 0$, and $\neq 0$) are "observed." However only three of them (< 0, ≤ 0, and $\neq 0$) are "observed to be true." [2]

These are natural properties to check and provide good coverage of a program's scalar values and control flow. This set is by no means complete, however; in particular, we believe it would be useful to have predicates on heap structures as well (see Section 4).

3. Cause Isolation Algorithm

This section presents our algorithm for automatically isolating multiple bugs. As discussed in Section 2, the input is a set of feedback reports from individual program runs *R*, where $R(P) = 1$ if predicate *P* is observed to be true during the execution of *R*.

The idea behind the algorithm is to simulate the iterative manner in which human programmers typically find and fix bugs:

1. Identify the most important bug *B*.
2. Fix *B*, and repeat.

For our purposes, identifying a bug *B* means selecting a predicate *P* closely correlated with its bug profile $\mathcal{B}$. The difficulty is that we know the set of runs that succeed and fail, but we do not know which set of failing runs corresponds to *B*, or even how many bugs there are. In other words, we do not know the sizes or membership of the set of bug profiles $\{\mathcal{B}_i\}$. Thus, in the first step we must infer which predicates are most likely to correspond to individual bugs and rank those predicates in importance.

For the second step, while we cannot literally fix the bug corresponding to the chosen predictor *P*, we can simulate what happens if the bug does not occur. We discard any run *R* such that $R(P) = 1$ and recursively apply the entire algorithm to the remaining runs. Discarding all the runs where $R(P) = 1$ reduces the importance of other predictors of *B*, allowing predicates that predict different bugs (i.e., corresponding to different sets of failing runs) to rise to the top in subsequent iterations.

3.1 Increase Scores

We now discuss the first step: how to find the cause of the most important bug. We break this step into two sub-steps. First, we eliminate predicates that have no predictive power at all; this typically reduces the number of predicates we need to consider by two orders of magnitude (e.g., from hundreds of thousands to thousands). Next, we rank the surviving predicates by importance (see Section 3.3).

Consider the following C code fragment:

```
f = ...;              (a)
if (f == NULL) {      (b)
    x = 0;            (c)
    *f;               (d)
}
```

Consider the predicate `f == NULL` at line `(b)`, which would be captured by branches instrumentation. Clearly this predicate is highly correlated with failure; in fact, whenever it is true this program inevitably crashes.[3] An important observation, however, is that there is no one perfect predictor of failure in a program with multiple bugs. Even a "smoking gun" such as `f == NULL` at line `(b)` has little or no predictive power for failures due to unrelated bugs in the same program.

The bug in the code fragment above is *deterministic* with respect to `f == NULL`: if `f == NULL` is true at line `(b)`, the program fails. In many cases it is impossible to observe the exact conditions causing failure; for example, buffer overrun bugs in a C program may or may not cause the program to crash depending on runtime system decisions about how data is laid out in memory. Such bugs are *non-deterministic* with respect to every observed predicate; even for the best predictor *P*, it is possible that *P* is true and still the program terminates normally. In the example above, if we insert before line `(d)` a valid pointer assignment to `f` controlled by a conditional that is true at least occasionally (say via a call to read input)

```
if (...) f = ... some valid pointer ...;
*f;
```

the bug becomes non-deterministic with respect to `f == NULL`.

To summarize, even for a predicate *P* that is truly the cause of a bug, we can neither assume that when *P* is true that the program fails nor that when *P* is never observed to be true that the program succeeds. But we can express the probability that *P* being true implies failure. Let *Crash* be an atomic predicate that is true for failing runs and false for successful runs. Let $Pr(A|B)$ denote the conditional probability function of the event *A* given event *B*. We want to compute:

$$\mathit{Failure}(P) \equiv \mathit{Pr}(\mathit{Crash}|P \text{ observed to be true})$$

for every instrumented predicate *P* over the set of all runs. Let $S(P)$ be the number of successful runs in which *P* is observed to be true, and let $F(P)$ be the number of failing runs in which *P* is observed to be true. We estimate $\mathit{Failure}(P)$ as:

$$\mathit{Failure}(P) = \frac{F(P)}{S(P) + F(P)}$$

---

[2] In reality, we count the number of times *P* is observed to be true, but the analysis of the feedback reports only uses whether *P* is observed to be true at least once.

[3] We also note that this bug could be detected by a simple static analysis; this example is meant to be concise rather than a significant application of our techniques.

Notice that *Failure*(P) is unaffected by the set of runs in which P is not observed to be true. Thus, if P is the cause of a bug, the causes of other independent bugs do not affect *Failure*(P). Also note that runs in which P is not observed at all (either because the line of code on which P is checked is not reached, or the line is reached but P is not sampled) have no effect on *Failure*(P). The definition of *Failure*(P) generalizes the idea of deterministic and non-deterministic bugs. A bug is deterministic for P if *Failure*$(P) =$ 1.0, or equivalently, P is never observed to be true in a successful run ($S(P) = 0$) and P is observed to be true in at least one failing run ($F(P) > 0$). If *Failure*$(P) < 1.0$ then the bug is non-deterministic with respect to P. Lower scores show weaker correlation between the predicate and program failure.

Now *Failure*(P) is a useful measure, but it is not good enough for the first step of our algorithm. To see this, consider again the code fragment given above in its original, deterministic form. At line `(b)` we have *Failure*(`f == NULL`) $= 1.0$, so this predicate is a good candidate for the cause of the bug. But on line `(c)` we have the unpleasant fact that *Failure*(`x == 0`) $= 1.0$ as well. To understand why, observe that the predicate `x == 0` is always true at line `(c)` and, in addition, only failing runs reach this line. Thus S(`x == 0`) $= 0$, and, so long as there is at least one run that reaches line `(c)` at all, *Failure*(`x == 0`) at line `(c)` is 1.0.

As this example shows, just because *Failure*(P) is high does not mean P is the cause of a bug. In the case of `x == 0`, the decision that eventually causes the crash is made earlier, and the high *Failure*(`x == 0`) score merely reflects the fact that this predicate is checked on a path where the program is already doomed.

A way to address this difficulty is to score a predicate not by the chance that it implies failure, but by how much difference it makes that the predicate is observed to be true versus simply reaching the line where the predicate is checked. That is, on line `(c)`, the probability of crashing is already 1.0 regardless of the value of the predicate `x == 0`, and thus the fact that `x == 0` is true does not increase the probability of failure at all. This fact coincides with our intuition that this predicate is irrelevant to the bug.

Recall that we write "P observed" when P has been reached and sampled at least once, without regard to whether P was actually true or false. This leads us to the following definition:

$$Context(P) \equiv Pr(Crash|P \text{ observed})$$

Now, it is not the case that P is observed in every run, because the site where this predicate occurs may not be reached, or may be reached but not sampled. Thus, *Context*(P) is the probability that in the subset of runs where the site containing predicate P is reached and sampled, the program fails. We can estimate *Context*(P) as follows:

$$Context(P) = \frac{F(P \text{ observed})}{S(P \text{ observed}) + F(P \text{ observed})}$$

The interesting quantity, then, is

$$Increase(P) \equiv Failure(P) - Context(P)$$

which can be read as: How much does P being true increase the probability of failure over simply reaching the line where P is sampled? For example, for the predicate `x == 0` on line `(c)`, we have

$$Failure(\texttt{x == 0}) = Context(\texttt{x == 0}) = 1.0$$

and so *Increase*(`x == 0`) $= 0$.

In most cases, a predicate P with *Increase*$(P) \leq 0$ has no predictive power and can safely be discarded. (See Section 5 for possible exceptions.) Because some *Increase*(P) scores may be based on few observations of P, it is important to attach confidence intervals to the scores. In our experiments we retain a predicate P only if the 95% confidence interval based on *Increase*(P) lies strictly above zero; this removes predicates from consideration that have high increase scores but very low confidence because of few observations.

Pruning predicates based on *Increase*(P) has several desirable properties. It is easy to prove that large classes of irrelevant predicates always have scores ≤ 0. For example, any predicate that is unreachable, that is a program invariant, or that is obviously control-dependent on a true cause is eliminated by this test. It is also worth pointing out that this test tends to localize bugs at a point where the condition that causes the bug first becomes true, rather than at the crash site. For example, in the code fragment given above, the bug is attributed to the success of the conditional branch test `f == NULL` on line `(b)` rather than the pointer dereference on line `(d)`. Thus, the cause of the bug discovered by the algorithm points directly to the conditions under which the crash occurs, rather than the line on which it occurs (which is usually available anyway in the stack trace).

3.2 Statistical Interpretation

We have explained the test *Increase*$(P) > 0$ using programming terminology, but it also has a natural statistical interpretation as a simplified likelihood ratio hypothesis test. Consider the two classes of trial runs of the program: failed runs F and successful runs S. For each class, we can treat the predicate P as a Bernoulli random variable with heads probabilities $\pi_f(P)$ and $\pi_s(P)$, respectively, for the two classes. The heads probability is the probability that the predicate is observed to be true. If a predicate causes a set of crashes, then π_f should be much bigger than π_s. We can formulate two statistical hypotheses: the null hypothesis $\mathcal{H}_0 : \pi_f \leq \pi_s$, versus the alternate hypothesis $\mathcal{H}_1 : \pi_f > \pi_s$. Since π_f and π_s are not known, we must estimate them:

$$\hat{\pi}_f(P) = \frac{F(P)}{F(P \text{ observed})} \qquad \hat{\pi}_s(P) = \frac{S(P)}{S(P \text{ observed})}$$

Although these proportion estimates of π_f and π_s approach the actual heads probabilities as we increase the number of trial runs, they still differ due to sampling. With a certain probability, using these estimates instead of the actual values results in the wrong answer. A *likelihood ratio test* takes this uncertainty into account, and makes use of the statistic $Z = \frac{(\hat{\pi}_f - \hat{\pi}_s)}{V_{f,s}}$, where $V_{f,s}$ is a sample variance term [8]. When the data size is large, Z can be approximated as a standard Gaussian random variable. Performed independently for each predicate P, the test decides whether or not $\pi_f(P) \leq \pi_s(P)$ with a guaranteed false-positive probability (i.e., choosing $\mathcal{H}_1$ when $\mathcal{H}_0$ is true). A necessary (but not sufficient) condition for choosing $\mathcal{H}_1$ is that $\hat{\pi}_f(P) > \hat{\pi}_s(P)$. However, this is equivalent to the condition that *Increase*$(P) > 0$. To see why, let $a = F(P)$, $b = S(P)$, $c = F(P \text{ observed})$, and $d = S(P \text{ observed})$. Then

$$\begin{aligned} &Increase(P) > 0 \iff Failure(P) > Context(P) \\ \iff &\frac{a}{a+b} > \frac{c}{c+d} \iff a(c+d) > (a+b)c \\ \iff &ad > bc \iff \frac{a}{c} > \frac{b}{d} \iff \hat{\pi}_f(P) > \hat{\pi}_s(P) \end{aligned}$$

3.3 Balancing Specificity and Sensitivity

We now turn to the question of ranking those predicates that survive pruning. Table 1 shows the top predicates under different ranking schemes (explained below) for one of our experiments. Due to space limitations we omit additional per-predicate information, such as source file and line number, which is available in the interactive version of our analysis tools.

We use a concise *bug thermometer* to visualize the information for each predicate. The length of the thermometer is logarithmic in the number of runs in which the predicate was observed, so

Table 1. Comparison of ranking strategies for MOSS without redundancy elimination

(a) Sort descending by F(P)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.007 ± 0.012	22554	5045	27599	`files[filesindex].language != 15`
	0.176	0.007 ± 0.012	22566	5045	27611	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.012	22571	5045	27616	`strcmp != 0`
	0.176	0.007 ± 0.013	18894	4251	23145	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.013	18885	4240	23125	`files[filesindex].language != 14`
	0.176	0.008 ± 0.013	17757	4007	21764	`filesindex >= 25`
	0.177	0.008 ± 0.014	16453	3731	20184	`new value of M < old value of M`
	0.176	0.261 ± 0.023	4800	3716	8516	`config.winnowing_window_size != argc`
.... 2732 additional predictors follow						

(b) Sort descending by Increase(P)

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.065	0.935 ± 0.019	0	23	23	`((*(fi + i)))->this.last_token < filesbase`
	0.065	0.935 ± 0.020	0	10	10	`((*(fi + i)))->other.last_line == last`
	0.071	0.929 ± 0.020	0	18	18	`((*(fi + i)))->other.last_line == filesbase`
	0.073	0.927 ± 0.020	0	10	10	`((*(fi + i)))->other.last_line == yy_n_chars`
	0.071	0.929 ± 0.028	0	19	19	`bytes <= filesbase`
	0.075	0.925 ± 0.022	0	14	14	`((*(fi + i)))->other.first_line == 2`
	0.076	0.924 ± 0.022	0	12	12	`((*(fi + i)))->this.first_line < nid`
	0.077	0.923 ± 0.023	0	10	10	`((*(fi + i)))->other.last_line == yy_init`
.... 2732 additional predictors follow						

(c) Sort descending by harmonic mean

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.824 ± 0.009	0	1585	1585	`files[filesindex].language > 16`
	0.176	0.824 ± 0.009	0	1584	1584	`strcmp > 0`
	0.176	0.824 ± 0.009	0	1580	1580	`strcmp == 0`
	0.176	0.824 ± 0.009	0	1577	1577	`files[filesindex].language == 17`
	0.176	0.824 ± 0.009	0	1576	1576	`tmp == 0 is TRUE`
	0.176	0.824 ± 0.009	0	1573	1573	`strcmp > 0`
	0.116	0.883 ± 0.012	1	774	775	`((*(fi + i)))->this.last_line == 1`
	0.116	0.883 ± 0.012	1	776	777	`((*(fi + i)))->other.last_line == yyleng`
.... 2732 additional predictors follow						

small increases in thermometer size indicate many more runs. Each thermometer has a sequence of bands. The black band on the left shows *Context*(P) as a fraction of the entire thermometer length. The dark gray band shows the lower bound of *Increase*(P) with 95% confidence, also proportional to the entire thermometer length. The light gray band shows the size of that confidence interval.[4] It is very small in most thermometers, indicating a tight interval. The white space at the right end of the thermometer shows $S(P)$, the number of successful runs in which the predicate was observed to be true. The tables show the thermometer as well as the numbers for each of the quantities that make up the thermometer.

The most important bug is the one that causes the greatest number of failed runs. This observation suggests:

$$\mathit{Importance}(P) = F(P)$$

Table 1(a) shows the top predicates ranked by decreasing $F(P)$ after predicates where $\mathit{Increase}(P) \leq 0$ are discarded. The predicates in Table 1(a) are, as expected, involved in many failing runs. However, the large white band in each thermometer reveals that these predicates are also highly non-deterministic: they are also true in many successful runs and are weakly correlated with bugs. Furthermore, the very narrow dark gray bands in most thermometers indicate that most *Increase* scores are very small.

Our experience with other ranking strategies that emphasize the number of failed runs is similar. They select predicates involved in many failing, but also many successful, runs. The best of these predicates (the ones with high *Increase* scores) are *super-bug predictors*: predictors that include failures from more than one bug. Super-bug predictors account for a very large number of failures by combining the failures of multiple bugs, but are also highly non-deterministic (because they are not specific to any single cause of failure) despite reasonably high *Increase* scores.

Another possibility is:

$$\mathit{Importance}(P) = \mathit{Increase}(P)$$

Table 1(b) shows the top predicates ranked by decreasing *Increase* score. Thermometers here are almost entirely dark gray, indicating *Increase* scores that are very close to 1.0. These predicates do a much better job of predicting failure. In fact, the program always fails when any of these predicates is true. However, observe that the number of failing runs (column F) is very small. These predicates are *sub-bug predictors*: predictors for a subset of the failures caused by a bug. Unlike super-bug predictors, which are not useful in our experience, sub-bug predictors that account for a significant fraction of the failures for a bug often provide valu-

[4] If reading this paper in color, the dark gray band is red, and the light gray band is pink.

able clues. However still they represent special cases and may not suggest other, more fundamental, causes of the bug.

Tables 1(a) and 1(b) illustrate the difficulty of defining "importance." We are looking for predicates with high *sensitivity*, meaning predicates that account for many failed runs. But we also want high *specificity*, meaning predicates that do not mis-predict failure in many successful runs. In information retrieval, the corresponding terms are *recall* and *precision*. A standard way to combine sensitivity and specificity is to compute their harmonic mean; this measure prefers high scores in both dimensions. In our case, $Increase(P)$ measures specificity. For sensitivity, we have found it useful to consider a transformation ϕ of the raw counts, and to form the normalized ratio $\phi(F(P))/\phi(NumF)$, where *NumF* is the total number of failed runs. In our work thus far ϕ has been a logarithmic transformation, which moderates the impact of very large numbers of failures. Thus our overall metric is the following:

$$Importance(P) = \frac{2}{\frac{1}{Increase(P)} + \frac{1}{log(F(P))/log(NumF)}}$$

It is possible for this formula to be undefined (due to a division by 0), in which case we define the *Importance* to be 0. Table 1(c) gives results using this metric. Both individual $F(P)$ counts and individual $Increase(P)$ scores are smaller than in Tables 1(a) and 1(b), but the harmonic mean has effectively balanced both of these important factors. All of the predicates on this list indeed have both high specificity and sensitivity. Each of these predictors accurately describes a large number of failures.

As in the case of the pruning based on the *Increase* score, it is useful to assess statistical significance of *Importance* scores by computing a confidence interval for the harmonic mean. Exact confidence intervals for the harmonic mean are not available, but we can use the delta method to derive approximate confidence intervals [9]. Computing the means and variances needed for applying the delta method requires computing estimates of underlying binomial probabilities for the predicates and conditioning on the event that the corresponding counts are non-zero.

3.4 Redundancy Elimination

The remaining problem with the results in Table 1(c) is that there is substantial redundancy; it is easy to see that several of these predicates are related. This redundancy hides other, distinct bugs that either have fewer failed runs or more non-deterministic predictors further down the list. As discussed previously, beginning with the set of all runs and predicates, we use a simple iterative algorithm to eliminate redundant predicates:

1. Rank predicates by *Importance*.
2. Remove the top-ranked predicate P and discard all runs R (feedback reports) where $R(P) = 1$.
3. Repeat these steps until the set of runs is empty or the set of predicates is empty.

We can now state an easy-to-prove but important property of this algorithm.

LEMMA 3.1. *Let $P_1, \ldots, P_n$ be a set of instrumented predicates, $B_1, \ldots, B_m$ a set of bugs, and $\mathcal{B}_1, \ldots, \mathcal{B}_m$ the corresponding bug profiles. Let*

$$\mathcal{Z} = \bigcup_{1 \leq i \leq n} \{R | R(P_i) = 1\}.$$

If for all $1 \leq j \leq m$ we have $\mathcal{B}_j \cap \mathcal{Z} \neq \emptyset$, then the algorithm chooses at least one predicate from the list $P_1, \ldots, P_n$ that predicts at least one failure due to B_j.

Thus, the elimination algorithm chooses at least one predicate predictive of each bug represented by the input set of predicates. We are, in effect, covering the set of bugs with a ranked subset of predicates. The other property we might like, that the algorithm chooses exactly one predicate to represent each bug, does not hold; we shall see in Section 4 that the algorithm sometimes selects a strong sub-bug predictor as well as a more natural predictor. (As a technical aside, note that Lemma 3.1 does not guarantee that every selected predicate has a positive *Increase* score at the time it is selected. Even if predicates with non-positive *Increase* scores are discarded before running the elimination algorithm, new ones can arise during the elimination algorithm. However, the *Increase* score of any predicate that covers at least one failing run will at least be defined. See Section 5 for related discussion.)

Beyond always representing each bug, the algorithm works well for two other reasons. First, two predicates are redundant if they predict the same (or nearly the same) set of failing runs. Thus, simply removing the set of runs in which a predicate is true automatically reduces the importance of any related predicates in the correct proportions. Second, because elimination is iterative, it is only necessary that *Importance* selects a good predictor at each step, and not necessarily the best one; any predicate that covers a different set of failing runs than all higher-ranked predicates is selected eventually.

Finally, we studied an optimization in which we eliminated logically redundant predicates within instrumentation sites prior to running the iterative algorithm. However, the elimination algorithm proved to be sufficiently powerful that we obtained nearly identical experimental results with and without this optimization, indicating it is unnecessary.

4. Experiments

In this section we present the results of applying the algorithm described in Section 3 in five case studies. Table 2 shows summary statistics for each of the experiments. In each study we ran the programs on about 32,000 random inputs. The number of instrumentation sites varies with the size of the program, as does the number of predicates those instrumentation sites yield. Our algorithm is very effective in reducing the number of predicates the user must examine. For example, in the case of RHYTHMBOX an initial set of 857,384 predicates is reduced to 537 by the $Increase(P) > 0$ test, a reduction of 99.9%. The elimination algorithm then yields 15 predicates, a further reduction of 97%. The other case studies show a similar reduction in the number of predicates by 3-4 orders of magnitude.

The results we discuss are all on sampled data. Sampling creates additional challenges that must be faced by our algorithm. Assume P_1 and P_2 are equivalent bug predictors and both are sampled at a rate of $1/100$ and both are reached once per run. Then even though P_1 and P_2 are equivalent, they will be observed in nearly disjoint sets of runs and treated as close to independent by the elimination algorithm.

To address this problem, we set the sampling rates of predicates to be inversely proportional to their frequency of execution. Based on a training set of 1,000 executions, we set the sampling rate of each predicate so as to obtain an expected 100 samples of each predicate in subsequent program executions. On the low end, the sampling rate is clamped to a minimum of $1/100$; if the site is expected to be reached fewer than 100 times the sampling rate is set at 1.0. Thus, rarely executed code has a much higher sampling rate than very frequently executed code. (A similar strategy has been pursued for similar reasons in related work [3].) We have validated this approach by comparing the results for each experiment with results obtained with no sampling at all (i.e., the sampling rate of all predicates set to 100%). The results are identical except for the RHYTHMBOX and MOSS experiments, where we judge the differences to be minor: sometimes a different but logically equivalent predicate is chosen, the ranking of predictors of different

Table 2. Summary statistics for bug isolation experiments

		Runs			Predicate Counts		
	Lines of Code	Successful	Failing	Sites	Initial	*Increase* > 0	Elimination
MOSS	6001	26,299	5598	35,223	202,998	2740	21
CCRYPT	5276	20,684	10,316	9948	58,720	50	2
BC	14,288	23,198	7802	50,171	298,482	147	2
EXIF	10,588	30,789	2211	27,380	156,476	272	3
RHYTHMBOX	56,484	12,530	19,431	14,5176	857,384	537	15

bugs is slightly different, or one or the other version has a few extra, weak predictors at the tail end of the list.

4.1 A Validation Experiment

To validate our algorithm we first performed an experiment in which we knew the set of bugs in advance. We added nine bugs to MOSS, a widely used service for detecting plagiarism in software [15]. Six of these were previously discovered and repaired bugs in MOSS that we reintroduced. The other three were variations on three of the original bugs, to see if our algorithm could discriminate between pairs of bugs with very similar behavior but distinct causes. The nature of the eight crashing bugs varies: four buffer overruns, a null file pointer dereference in certain cases, a missing end-of-list check in the traversal of a hash table bucket, a missing out-of-memory check, and a violation of a subtle invariant that must be maintained between two parts of a complex data structure. In addition, some of these bugs are non-deterministic and may not even crash when they should.

The ninth bug—incorrect comment handling in some cases—only causes incorrect output, not a crash. We include this bug in our experiment in order to show that bugs other than crashing bugs can also be isolated using our techniques, provided there is some way, whether by automatic self-checking or human inspection, to recognize failing runs. In particular, for our experiment we also ran a correct version of MOSS and compared the output of the two versions. This oracle provides a labeling of runs as "success" or "failure," and the resulting labels are treated identically by our algorithm as those based on program crashes.

Table 3 shows the results of the experiment. The predicates listed were selected by the elimination algorithm in the order shown. The first column is the initial bug thermometer for each predicate, showing the *Context* and *Increase* scores before elimination is performed. The second column is the *effective* bug thermometer, showing the *Context* and *Increase* scores for a predicate P at the time P is selected (i.e., when it is the top-ranked predicate). Thus the effective thermometer reflects the cumulative diluting effect of redundancy elimination for all predicates selected before this one.

As part of the experiment we separately recorded the exact set of bugs that actually occurred in each run. The columns at the far right of Table 3 show, for each selected predicate and for each bug, the actual number of failing runs in which both the selected predicate is observed to be true and the bug occurs. Note that while each predicate has a very strong spike at one bug, indicating it is a strong predictor of that bug, there are always some runs with other bugs present. For example, the top-ranked predicate, which is overwhelmingly a predictor of bug #5, also includes some runs where bugs #3, #4, and #9 occurred. This situation is not the result of misclassification of failing runs by our algorithm. As observed in Section 1, more than one bug may occur in a run. It simply happens that in some runs bugs #5 and #3 both occur (to pick just one possible combination).

A particularly interesting case of this phenomenon is bug #7, one of the buffer overruns. Bug #7 is not strongly predicted by any predicate on the list but occurs in at least a few of the failing runs of most predicates. We have examined the runs of bug #7 in detail and found that the failing runs involving bug #7 also trigger at least one other bug. That is, even when the bug #7 overrun occurs, it never causes incorrect output or a crash in any run. Bug #8, another overrun, was originally found by a code inspection. It is not shown here because the overrun is never triggered in our data (its column would be all 0's). There is no way our algorithm can find causes of bugs that do not occur, but recall that part of our purpose in sampling user executions is to get an accurate picture of the most important bugs. It is consistent with this goal that if a bug never causes a problem, it is not only not worth fixing, it is not even worth reporting.

The other bugs all have strong predictors on the list. In fact, the top eight predicates have exactly one predictor for each of the seven bugs that occur, with the exception of bug #1, which has one very strong sub-bug predictor in the second spot and another predictor in the sixth position. Notice that even the rarest bug, bug #2, which occurs more than an order of magnitude less frequently than the most common bug, is identified immediately after the last of the other bugs. Furthermore, we have verified by hand that the selected predicates would, in our judgment, lead an engineer to the cause of the bug. Overall, the elimination algorithm does an excellent job of listing separate causes of each of the bugs in order of priority, with very little redundancy.

Below the eighth position there are no new bugs to report and every predicate is correlated with predicates higher on the list. Even without the columns of numbers at the right it is easy to spot the eighth position as the natural cutoff. Keep in mind that the length of the thermometer is on a log scale, hence changes in larger magnitudes may appear less evident. Notice that the initial and effective thermometers for the first eight predicates are essentially identical. Only the predicate at position six is noticeably different, indicating that this predicate is somewhat affected by a predicate listed earlier (specifically, its companion sub-bug predictor at position two). However, all of the predicates below the eighth line have very different initial and effective thermometers (either many fewer failing runs, or much more non-deterministic, or both) showing that these predicates are strongly affected by higher-ranked predicates.

The visualizations presented thus far have a drawback illustrated by the MOSS experiment: It is not easy to identify the predicates to which a predicate is closely related. Such a feature would be useful in confirming whether two selected predicates represent different bugs or are in fact related to the same bug. We do have a measure of how strongly P implies another predicate P': How does removing the runs where $R(P) = 1$ affect the importance of P'? The more closely related P and P' are, the more P''s importance drops when P's failing runs are removed. In the interactive version of our analysis tools, each predicate P in the final, ranked list links to an *affinity list* of all predicates ranked by how much P causes their ranking score to decrease.

Table 3. MOSS failure predictors using nonuniform sampling

Initial	Effective	Predicate	Number of Failing Runs Also Exhibiting Bug #*n* #1	#2	#3	#4	#5	#6	#7	#9
		`files[filesindex].language > 16`	0	0	28	54	1585	0	0	68
		`((*(fi + i)))->this.last_line == 1`	774	0	17	0	0	0	18	2
		`token_index > 500`	31	0	16	711	0	0	0	47
		`(p + passage_index)->last_token <= filesbase`	28	2	508	0	0	0	1	29
		`__result == 0 is TRUE`	16	0	0	9	19	291	0	13
		`config.match_comment is TRUE`	791	2	23	1	0	5	11	41
		`i == yy_last_accepting_state`	55	0	21	0	0	3	7	769
		`new value of f < old value of f`	3	144	2	2	0	0	0	5
		`files[fileid].size < token_index`	31	0	10	633	0	0	0	40
		`passage_index == 293`	27	3	8	0	0	0	2	366
		`((*(fi + i)))->other.last_line == yyleng`	776	0	16	0	0	0	18	1
		`min_index == 64`	24	1	7	0	0	1	1	249
		`((*(fi + i)))->this.last_line == yy_start`	771	0	18	0	0	0	19	0
		`(passages + i)->fileid == 52`	24	0	477	14	24	0	1	14
		`passage_index == 25`	60	5	27	0	0	4	10	962
		`strcmp > 0`	0	0	28	54	1584	0	0	68
		`i > 500`	32	2	18	853	54	0	0	53
		`token_sequence[token_index].val >= 100`	1250	3	28	38	0	15	19	65
		`i == 50`	27	0	11	0	0	1	4	463
		`passage_index == 19`	59	5	28	0	0	4	10	958
		`bytes <= filesbase`	1	0	19	0	0	0	0	1

Table 4. Predictors for CCRYPT

Initial	Effective	Predicate
		`res == nl`
		`line <= outfile`

Table 5. Predictors for BC

Initial	Effective	Predicate
		`a_names < v_names`
		`old_count == 32`

4.2 Additional Experiments

We briefly report here on experiments with additional applications containing both known and unknown bugs. Complete analysis results for all experiments may be browsed interactively at <http://www.cs.wisc.edu/˜liblit/pldi-2005/>.

4.2.1 CCRYPT

We analyzed CCRYPT 1.2, which has a known input validation bug. The results are shown in Table 4. Our algorithm reports two predictors, both of which point directly to the single bug. It is easy to discover that the two predictors are for the same bug; the first predicate is listed first in the second predicate's affinity list, indicating the first predicate is a sub-bug predictor associated with the second predicate.

4.2.2 BC

GNU BC 1.06 has a previously reported buffer overrun. Our results are shown in Table 5. The outcome is the same as for CCRYPT: two predicates are retained by elimination, and the second predicate lists the first predicate at the top of its affinity list, indicating that the first predicate is a sub-bug predictor of the second. Both predicates point to the cause of the overrun. This bug causes a crash long after the overrun occurs and there is no useful information on the stack at the point of the crash to assist in isolating this bug.

4.2.3 EXIF

Table 6 shows results for EXIF 0.6.9, an open source image processing program. Each of the three predicates is a predictor of a distinct

Table 6. Predictors for EXIF

Initial	Effective	Predicate
		`i < 0`
		`maxlen > 1900`
		`o + s > buf_size is TRUE`

and previously unknown crashing bug. It took less than twenty minutes of work to find and verify the cause of each of the bugs using these predicates and the additional highly correlated predicates on their affinity lists. All bugs have been confirmed as valid by EXIF project developers.

To illustrate how statistical debugging is used in practice, we use the last of these three failure predictors as an example, and describe how it enabled us to effectively isolate the cause of one of the bugs. Failed runs exhibiting `o + s > buf_size` show the following unique stack trace at the point of termination:

```
main
  exif_data_save_data
    exif_data_save_data_content
      exif_data_save_data_content
        exif_data_save_data_entry
          exif_mnote_data_save
            exif_mnote_data_canon_save
              memcpy
```

The code in the vicinity of the call to `memcpy` in function `exif_mnote_data_canon_save` is as follows:

```
for (i = 0; i < n->count; i++) {
    ...
    memcpy(*buf + doff,                    (c)
           n->entries[i].data, s);
    ...
}
```

This stack trace alone provides little insight into the cause of the bug. However, our algorithm highlights `o + s > buf_size` in function `exif_mnote_data_canon_load` as a strong bug predictor. Thus, a quick inspection of the source code leads us to construct the following call sequence:

Table 7. Predictors for RHYTHMBOX

Initial	Effective	Predicate
		`tmp is FALSE`
		`(mp->priv)->timer is FALSE`
		`(view->priv)->change_sig_queued is TRUE`
		`(hist->priv)->db is TRUE`
		`rb_playlist_manager_signals[0] > 269`
		`(db->priv)->thread_reaper_id >= 12`
		`entry == entry`
		`fn == fn`
		`klass > klass`
		`genre < artist`
		`vol <= (float )0 is TRUE`
		`(player->priv)->handling_error is TRUE`
		`(statusbar->priv)->library_busy is TRUE`
		`shell < shell`
		`len < 270`

```
main
  exif_loader_get_data
    exif_data_load_data
      exif_mnote_data_canon_load
  exif_data_save_data
    exif_data_save_data_content
      exif_data_save_data_content
        exif_data_save_data_entry
          exif_mnote_data_save
            exif_mnote_data_canon_save
              memcpy
```

The code in the vicinity of the predicate `o + s > buf_size` in function `exif_mnote_data_canon_load` is as follows:

```
for (i = 0; i < c; i++) {
    ...
    n->count = i + 1;
    ...
    if (o + s > buf_size) return;     (a)
    ...
    n->entries[i].data = malloc(s);   (b)
    ...
}
```

It is apparent from the above code snippets and the call sequence that whenever the predicate `o + s > buf_size` is true,

- the function `exif_mnote_data_canon_load` returns on line `(a)`, thereby skipping the call to `malloc` on line `(b)` and thus leaving `n->entries[i].data` uninitialized for some value of `i`, and
- the function `exif_mnote_data_canon_save` passes the uninitialized `n->entries[i].data` to `memcpy` on line `(c)`, which reads it and eventually crashes.

In summary, our algorithm enabled us to effectively isolate the causes of several previously unknown bugs in source code unfamiliar to us in a small amount of time and without any explicit specification beyond "the program shouldn't crash."

4.2.4 RHYTHMBOX

Table 7 shows our results for RHYTHMBOX 0.6.5, an interactive, graphical, open source music player. RHYTHMBOX is a complex, multi-threaded, event-driven system, written using a library providing object-oriented primitives in C. Event-driven systems use event queues; each event performs some computation and possibly adds more events to some queues. We know of no static analysis today that can analyze event-driven systems accurately, because no static analysis is currently capable of analyzing the heap-allocated event queues with sufficient precision. Stack inspection is also of limited utility in analyzing event-driven systems, as the stack in the main event loop is unchanging and all of the interesting state is in the queues.

We isolated two distinct bugs in RHYTHMBOX. The first predicate led us to the discovery of a race condition. The second predicate was not useful directly, but we were able to isolate the bug using the predicates in its affinity list. This second bug revealed what turned out to be a very common incorrect pattern of accessing the underlying object library (recall Section 1). RHYTHMBOX developers confirmed the bugs and enthusiastically applied patches within a few days, in part because we could quantify the bugs as important crashing bugs. It required several hours to isolate each of the two bugs (and there are additional bugs represented in the predictors that we did not isolate) because they were violations of subtle heap invariants that are not directly captured by our current instrumentation schemes. Note, however, that we could not have even begun to understand these bugs without the information provided by our tool. We intend to explore schemes that track predicates on heap structure in future work.

4.3 How Many Runs Are Needed?

Recall that we used about 32,000 runs in each of the five case studies. For many of the bugs this number is clearly far more than the minimum required. In this section, we estimate how many runs are actually needed for all bug predictors to be identified.

Our estimates are computed using the following methodology. We choose one predictor for each bug identified in the case studies. Where the elimination algorithm selects two predictors for a bug, we pick the more natural one (i.e., not the sub-bug predictor). For each chosen predictor P, we compute the importance of P using many different numbers of runs. Let $\mathit{Importance}_N(P)$ be the importance of P using N runs. We are interested in the minimum N such that

$$\mathit{Importance}_{32,000}(P) - \mathit{Importance}_N(P) < 0.2$$

The threshold 0.2 is selected because we observe that all of the chosen predictors in our studies would still be ranked very highly even if their importance scores were 0.2 lower.

Table 8 presents the results of the analysis. In these experiments, the number of runs N ranges over the values 100, 200, ..., 900, 1,000, 2,000, ..., 25,000. For each study, we list two numbers for each bug with predictor P: the minimum number of runs N such that the threshold test is met, and the number of failing runs $F(P)$ among those N runs where P is observed to be true. Note that the number of runs N needed for different bugs varies by two orders of magnitude. We need 21,000 runs to isolate all of the bug predictors in EXIF because the last bug in that study is extremely rare: only 21 failing runs out of our total population of 33,000 runs share bug #3 as the cause of failure. If we exclude bug #3, then just 2,000 runs are sufficient to isolate EXIF bugs #1 and #2. Thus, results degrade gracefully with fewer runs, with the predictors for rare bugs dropping out first.

The number $F(P)$ is independent of the rate at which the different bugs occur and allows us to compare the absolute number of failures needed to isolate different bugs. Notice that we can isolate any bug predictor with between 10 and 40 observations of failing runs caused by the bug. How long it takes to get those observations of failing runs depends on the frequency with which the bug occurs and the sampling rate of the bug predictor. Assume F failures are needed to isolate the predictor of a bug. If the failing runs where the predictor is observed to be true constitute a fraction $0 \leq p \leq 1$ of all runs, then about $N = F/p$ runs will be required by our algorithm.

4.4 Comparison with Logistic Regression

In earlier work we used ℓ_1-regularized logistic regression to rank the predicates by their failure-prediction strength [10, 16]. Logistic regression uses linearly weighted combinations of predicates to classify a trial run as successful or failed. Regularized logistic

Table 8. Minimum number of runs needed

	Runs	Bug #n						
		#1	#2	#3	#4	#5	#6	#9
MOSS	$F(P)$	18	10	32	12	21	11	20
	N	500	3,000	2,000	800	300	1,000	600
CCRYPT	$F(P)$	26						
	N	200						
BC	$F(P)$	40						
	N	200						
RHYTHMBOX	$F(P)$	22	35					
	N	300	100					
EXIF	$F(P)$	28	12	13				
	N	2,000	300	21,000				

Table 9. Results of logistic regression for MOSS

Coefficient	Predicate
0.769379	`(p + passage_index)->last_line < 4`
0.686149	`(p + passage_index)->first_line < i`
0.675982	`i > 20`
0.671991	`i > 26`
0.619479	`(p + passage_index)->last_line < i`
0.600712	`i > 23`
0.591044	`(p + passage_index)->last_line == next`
0.567753	`i > 22`
0.544829	`i > 25`
0.536122	`i > 28`

regression incorporates a penalty term that drives most coefficients towards zero, thereby giving weights to only the most important predicates. The output is a set of coefficients for predicates giving the best overall prediction.

A weakness of logistic regression for our application is that it seeks to cover the set of failing runs without regard to the orthogonality of the selected predicates (i.e., whether they represent distinct bugs). This problem can be seen in Table 9, which gives the top ten predicates selected by logistic regression for MOSS. The striking fact is that all selected predicates are either sub-bug or super-bug predictors. The predicates beginning with `p + ...` are all sub-bug predictors of bug #1 (see Table 3). The predicates `i > ...` are super-bug predictors: `i` is the length of the command line and the predicates say program crashes are more likely for long command lines (recall Section 1).

The prevalence of super-bug predictors on the list shows the difficulty of making use of the penalty term. Limiting the number of predicates that can be selected via a penalty has the effect of encouraging regularized logistic regression to choose super-bug predictors, as these cover more failing runs at the expense of poorer predictive power compared to predictors of individual bugs. On the other hand, the sub-bug predictors are chosen based on their excellent prediction power of those small subsets of failed runs.

5. Alternatives and Extensions

While we have targeted our algorithm at finding bugs, there are other possible applications, and there are variations of the basic approach we have presented that may prove useful. In this section we briefly discuss some of these possibilities.

While we have focused on bug finding, the same ideas can be used to isolate predictors of any program event. For example, we could potentially look for early predictors of when the program will raise an exception, send a message on the network, write to disk, or suspend itself. Furthermore, it is interesting to consider applications in which the predictors are used on-line by the running program; for example, knowing that a strong predictor of program failure has become true may enable preemptive action (see Section 6).

There are also variations on the specific algorithm we have proposed that are worth investigating. For example, we have chosen to discard all the runs where $R(P) = 1$ when P is selected by the elimination algorithm, but there are at least three natural choices:

1. When P is selected, discard all runs where $R(P) = 1$.
2. When P is selected, discard only failing runs where $R(P) = 1$.
3. When P is selected, relabel all failing runs where $R(P) = 1$ as successful runs.

We have already given the intuition for (1), our current choice. For (2), the idea is that whatever the bug is, it is not manifested in the successful runs and thus retaining all successful runs is more representative of correct program behavior. Proposal (3) goes one step further, asserting that even the failing runs will look mostly the same once the bug is fixed, and the best approximation to a program without the bug is simply that the failing runs are now successful runs.

On a more technical level, the three proposals differ in how much code coverage they preserve. By discarding no runs, proposal (3) preserves all the code paths that were executed in the original runs, while proposal (1) discards the most runs and so potentially renders more paths unreached in the runs that remain. This difference in paths preserved translates into differences in the *Failure* and *Context* scores of predicates under the different proposals. In fact, for a predicate P and its complement $\neg P$, it is possible to prove that just after predicate P is selected by the elimination algorithm, then

$$Increase_3(\neg P) \geq Increase_2(\neg P) \geq Increase_1(\neg P) = 0$$

where the subscripts indicate which proposal for discarding runs is used and assuming all the quantities are defined. Thus, proposal (1) is the most conservative, in the sense that only one of P or $\neg P$ can have positive predictive power, while proposal (3) potentially allows more predictors to have positive *Increase* scores.

This analysis reveals that a predicate P with a negative *Increase* score is not necessarily useless—the score may be negative only because it is temporarily overshadowed by stronger predictors of different bugs that are anti-correlated with P. It is possible to construct examples where both P and $\neg P$ are the best predictors of different bugs, but the result mentioned above assures us that once one is selected by the elimination algorithm, the other's *Increase* score is non-negative if it is defined. This line of reasoning also suggests that when using proposal (2) or (3) for discarding runs, a predicate P with *Increase*$(P) \leq 0$ should not be discarded in a preprocessing step, as P may become a positive predictor once $\neg P$ is selected by the elimination algorithm. In the case of proposal (1), only one of P or $\neg P$ can ever have a non-negative *Increase* score, and so it seems reasonable to eliminate predicates with negative scores early.

6. Related Work

In this section we briefly survey related work. There is currently a great deal of interest in applying static analysis to improve software quality. While we firmly believe in the use of static analysis to find and prevent bugs, our dynamic approach has advantages as well. A dynamic analysis can observe actual run-time values, which is often better than either making a very conservative static assumption about run-time values for the sake of soundness or allowing some very simple bugs to escape undetected. Another advantage of dynamic analysis, especially one that mines actual user executions for its data, is the ability to assign an accurate importance to each bug. Additionally, as we have shown, a dynamic analysis that does not require an explicit specification of the properties to check can find clues to a very wide range of errors, including classes of errors not considered in the design of the analysis.

The Daikon project [5] monitors instrumented applications to discover likely program invariants. It collects extensive trace information at run time and mines traces offline to accept or reject any of a wide variety of hypothesized candidate predicates. The DIDUCE project [7] tests a more restricted set of predicates within the client program, and attempts to relate state changes in candidate predicates to manifestation of bugs. Both projects assume complete monitoring, such as within a controlled test environment. Our goal is to use lightweight partial monitoring, suitable for either testing or deployment to end users.

Software tomography as realized through the GAMMA system [1, 12] shares our goal of low-overhead distributed monitoring of deployed code. GAMMA collects code coverage data to support a variety of code evolution tasks. Our instrumentation exposes a broader family of data- and control-dependent predicates on program behavior and uses randomized sparse sampling to control overhead. Our predicates do, however, also give coverage information: the sum of all predicate counters at a site reveals the relative coverage of that site.

Efforts to directly apply statistical modeling principles to debugging have met with mixed results. Early work in this area by Burnell and Horvitz [2] uses program slicing in conjunction with Bayesian belief networks to filter and rank the possible causes for a given bug. Empirical evaluation shows that the slicing component alone finds 65% of bug causes, while the probabilistic model correctly identifies another 10%. This additional payoff may seem small in light of the effort, measured in man-years, required to distill experts' often tacit knowledge into a formal belief network. However, the approach does illustrate one strategy for integrating information about program structure into the statistical modeling process.

In more recent work, Podgurski et al. [13] apply statistical feature selection, clustering, and multivariate visualization techniques to the task of classifying software failure reports. The intent is to bucket each report into an equivalence group believed to share the same underlying cause. Features are derived offline from fine-grained execution traces without sampling; this approach reduces the noise level of the data but greatly restricts the instrumentation schemes that are practical to deploy outside of a controlled testing environment. As in our own earlier work, Podgurski uses logistic regression to select features that are highly predictive of failure. Clustering tends to identify small, tight groups of runs that do share a single cause but that are not always maximal. That is, one cause may be split across several clusters. This problem is similar to covering a bug profile with sub-bug predictors.

In contrast, current industrial practice uses stack traces to cluster failure reports into equivalence classes. Two crash reports showing the same stack trace, or perhaps only the same top-of-stack function, are presumed to be two reports of the same failure. This heuristic works to the extent that a single cause corresponds to a single point of failure, but our experience with MOSS, RHYTHMBOX, and EXIF suggests that this assumption may not often hold. In MOSS, we find that only bugs #2 and #5 have truly unique "signature" stacks: a crash location that is present if and only if the corresponding bug was actually triggered. These bugs are also our most deterministic. Bugs #4 and #6 also have nearly unique stack signatures. The remaining bugs are much less consistent: each stack signature is observed after a variety of different bugs, and each triggered bug causes failure in a variety of different stack states. RHYTHMBOX and EXIF bugs caused crashes so long after the bad behavior that stacks were of limited use or no use at all.

Studies that attempt real-world deployment of monitored software must address a host of practical engineering concerns, from distribution to installation to user support to data collection and warehousing. Elbaum and Hardojo [4] have reported on a limited deployment of instrumented Pine binaries. Their experiences have helped to guide our own design of a wide public deployment of applications with sampled instrumentation, presently underway [11].

For some highly available systems, even a single failure must be avoided. Once the behaviors that predict imminent failure are known, automatic corrective measures may be able to prevent the failure from occurring at all. The Software Dependability Framework (SDF) [6] uses multivariate state estimation techniques to model and thereby predict impending system failures. Instrumentation is assumed to be complete and is typically domain-specific. Our algorithm could also be used to identify *early warning* predicates that predict impending failure in actual use.

7. Conclusions

We have demonstrated a practical, scalable algorithm for isolating multiple bugs in complex software systems. Our experimental results show that we can detect a wide variety of both anticipated and unanticipated causes of failure in realistic systems and do so with a relatively modest number of program executions.

References

[1] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In M. B. Dwyer, editor, *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE-02)*, volume 28, 1 of *SOFTWARE ENGINEERING NOTES*, pages 2–9. ACM Press, 2002.

[2] L. Burnell and E. Horvitz. Structure and chance: melding logic and probability for software debugging. *Communications of the ACM*, 38(3):31–41, 57, Mar. 1995.

[3] T. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS: Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 2004.

[4] S. Elbaum and M. Hardojo. Deploying instrumented software to assist the testing activity. In RAMSS 2003 [14], pages 31–33.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[6] K. C. Gross, S. McMaster, A. Porter, A. Urmanov, and L. G. Votta. Proactive system maintenance using software telemetry. In RAMSS 2003 [14], pages 24–26.

[7] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 291–301. ACM Press, 2002.

[8] E. Lehmann. *Testing Statistical Hypotheses*. John Wiley & Sons, 2nd edition, 1986.

[9] E. Lehmann and G. Casella. *Theory of Point Estimation*. Springer, 2nd edition, 2003.

[10] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In J. James B. Fenwick and C. Norris, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, volume 38, 5 of *ACM SIGPLAN Notices*, pages 141–154. ACM Press, 2003.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Public deployment of cooperative bug isolation. In *Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, pages 57–62, Edinburgh, Scotland, May 24 2004.

[12] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137. ACM Press, 2003.

[13] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 465–477. IEEE Computer Society, 2003.

[14] *RAMSS '03: The 1st International Workshop on Remote Analysis and Measurement of Software Systems*, May 2003.

[15] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In ACM, editor, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pages 76–85, New York, NY 10036, USA, 2003. ACM Press.

[16] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.