Contemplating Navel: Improving Software Error Reporting by Classifying Program Behavior

Jungwoo Ha Hany Ramadan Jason V. Davis Christopher J. Rossbach
Indrajit Roy Donald E. Porter Emmett Witchel

Department of Computer Sciences, The University of Texas at Austin
{habals,ramadan,jdavis,rossbach,indrajit,porterde,witchel}@cs.utexas.edu

Abstract

End-user software problems take too much time to resolve, in part due to unclear or ambiguous error messages. The quality of error messages embedded within software is unlikely to improve given the variety of contexts in which errors can occur, the programming complexity of sophisticated error reporting, and the modular structure of modern applications. While vendors supply documents, help systems and websites to support end users, it is still difficult for users to figure out how to resolve their problems.

Navel improves error reporting by monitoring software execution and determining if a particular execution is an instance of a known error. As a program executes, Navel builds a compact abstraction of the program's behavior (a behavior profile) using control flow information. Navel classifies behavior profiles using a machine learning model trained on known errors by vendors, support organizations or other users, enabling them to better disseminate error workarounds by matching user behavior profiles with known problems. Navel provides a way for an average user to get solutions to software problems with less effort.

A prototype implementation, Skepsis, demonstrates the efficacy of the Navel approach. Skepsis collects three behavior profiles based on program control flow: function counting, path profiling, and a new technique, call-tree profiling. We evaluate Skepsis on confusing error messages currently emitted by large, mature programs including the `gcc` compiler and Microsoft's Visual FoxPro database. Using call-tree profiling, Skepsis achieves an average classification accuracy of 97% across a range of nine benchmarks on two operating systems, while function counting and path profiling achieve average classification accuracies of 92% and 94% respectively.

Categories and Subject Descriptors D [*2*]: 5

General Terms Reliability, Verification, Experimentation

Keywords error reporting, profiling, software support, machine learning

Technical Report *TR-06-11*. Department of Computer Sciences, The University of Texas at Austin

1. Introduction

Bad error messages have been around as long as software. Despite decades of improvements in languages, static analysis [4, 11, 38], program verification [16], testing methodologies [42, 13], and development tools, bad error messages show no signs of abating. The cost of administering, configuring and updating a machine's software is surpassing the cost of its hardware [22], and a large portion of modern software support cost comes from time wasted with bad error messages. A bad error message is any message that does not provide sufficient information for a user to fix the problem in a timely fashion. One recent study concluded that up to 25 percent of a system administrator's time may be spent following blind alleys suggested by poorly constructed and unclear messages [6]. If IT professionals struggle to administer systems, home users probably fare worse, since they do not have the time and expertise to successfully diagnose many of their problems.

There are several reasons why error messages embedded in software are difficult to improve. First, since the text of the error messages are written at the same time as the code itself, the usefulness of the text is limited by the foresight of the developer. Second, modular programming style hinders the quality of error reporting. Modules are designed as building blocks to be re-used in multiple disparate contexts, so they often abstract away the precise context that makes a good error message possible. Third, communication between protection boundaries prohibits a good error message. Programs such as `iptables` and `iproute2` consist of user code and a kernel module, and the error reporting interface from the kernel to user space is constrained to a simple list of overloaded error codes.

Due to the limitations of program error reporting, software vendors provide other support channels such as help files, support websites, and user forums. However, end-users suffer from the arduous task of ferreting out the exact solution they need from this barrage of information. Delivering the correct and exact solution to the end-user is the crucial problem of error reporting system.

Navel is an error reporting system that classifies program behavior, enabling matching an erroneous execution with a solution already found by others. By framing the problem of software support in term of program behavior classification, Navel can provide diagnostic help for a wide range of symptoms, including unclear error messages, crashes, hangs, and poor performance. This paper focuses on improving error reporting.

Figure 1 shows the Navel process. A program is instrumented to produce a profile of its behavior (e.g., function call counts, or a path profile). A small groups of users and/or developers train a machine-learning classifier on the behavior profile by labeling error executions. The label is a solution to the underlying problem for that error case. Users can then use the classifier to get the solution for their own error behavior. While diagnosing a software problem today might involve typing the error text into a web search engine and winnowing the responses with human intelligence, Navel reduces the task of an average user finding the advice of the expert to running a classifier on a program profile. The residual complexity of running a classifier on a program profile can be hidden by a simple interface we call, “hitting the Navel button” when something goes wrong. Applications can be instrumented by users or software distributors. A software distributor would use Navel as a common support infrastructure for all of its applications or as a way to offload its support burden. Open source developers would use Navel to simplify the process by which the average user gains wisdom from more experienced users.

Navel is trained by a community of users. The *experts* are people who currently diagnose problems and develop work-arounds, and post their solutions to software support websites. They provide the improved error messages reported by Navel. *Informed* users search software support websites, and implement the solutions they read about. They provide the labeled executions used to train the Navel classifiers by giving feedback that they experienced the same problem that was identified by an expert (e.g., similar to the feedback form provided by some support websites). The *masses* can follow directions, but do not have the ability or interest to diagnose their computer problems. They use the Navel system. As the name implies, the masses vastly outnumber the experts and informed users. A small number of users train the system for a much larger group of non-expert users, and the system will continue to adapt as the support needs of the user community change. Navel takes the manual work already being done by a user community and makes it more accessible to the average user.

The contributions of this paper are:

- We present Navel, a design for a system that disambiguates different root causes of an error message by using a machine-learning model. Ours is the first such system to deal with this problem by monitoring program control flow. (Sections 3 and 5)
- We designed a new technique, *call-tree profiling*, that represents software behaviors more accurately, on average, than existing profiling techniques such as function counting, or path profiling. (Section 3.1)
- We evaluate a Navel prototype, Skepsis, on large, mature programs that currently produce unclear error messages and confusing behavior, such as `gcc` and Microsoft’s database program `FoxPro`. Our study is the most comprehensive in terms of number of benchmarks, and benchmark size and maturity for any system that builds a machine-learning model from program behavior. (Section 4)
- The machine-learning models built by Skepsis require human labeling effort. We present two novel methods to minimize the amount of human effort: unsupervised clustering of errors, and using a model trained for one version of the software on the next version without retraining. (Section 6)

In Section 2 we present the motivation for Navel and how a prototype would be used to improve error reporting. Section 3 describes how Navel abstracts program behavior. In Section 4 we evaluate Skepsis in terms of the accuracy of its models, and its runtime overhead. Section 5 examines the models built by Skepsis to understand how they can achieve such high accuracy. Section 6 describes how to reduce the amount of human effort with Navel. We position Navel with respect to related work in Section 7. We outline future work in Section 8 and conclude in Section 9.

2. Navel motivation and use

There is a natural temptation to believe that bad error reporting is a simple matter for programmers to fix. The peril of succumbing to

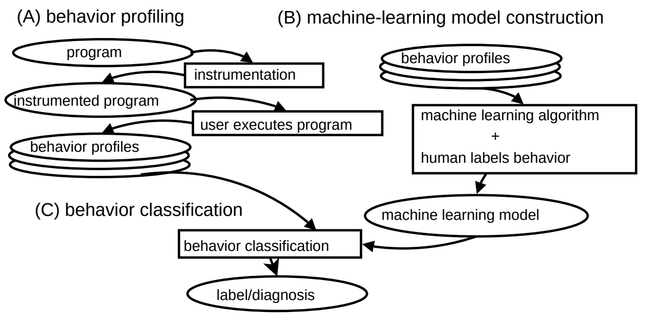

Figure 1. An overview of the Navel workflow. The rectangles represent processes consuming and producing data. Section A shows the steps that generate a behavior profile. Behavior profiles are labeled and used to generate machine-learning models (B), which are ultimately used to diagnose behaviors for users (C).

such temptation is to require each software project individually to expend scarce developer resources on error reporting. A common error reporting framework is more efficient and desirable. Navel is a common error reporting system that can be used by unmodified program binaries.

Complexity is the bane of software development, and reporting useful error messages is a complex problem. The problem space of what can possibly go wrong with a program is large, so testing all possibilities is extremely difficult. Because only a small subset of what could go wrong actually does, clarifying the error reports that actually occur in a software deployment can minimize the work required for error reporting while providing the most benefit to users.

Instead of burdening the application developer with the task of properly correlating application behavior with a proper error report, Navel correlates error reports with low-level details of application behavior that can be collected automatically. Sometimes a program can easily report a clear error, but as the following example, the example in Section 5.2, and the variety of cases in Table 3 demonstrate, this is not always the case for a variety of reasons. A solution beyond better programming is required.

2.1 NFS mount: when good error reporting goes bad

Using Linux 2.6.12, an unsuccessful attempt to mount an NFS file system can result in this error message.

```
NFSv3 not supported!
mount: wrong fs type, bad option,
        bad superblock on hostname:/tmp,
       missing codepage or other error
       In some cases useful info is found
        in syslog - try
       dmesg | tail  or so
```

The program output is actually two error messages from two locations in the mount code: the first location prints the message that NFSv3 is not supported, the second prints the verbose message with the suggestion to run `dmesg`. The server supports NFSv3, and there is no useful information in the syslog. What went wrong?

Everything after the first line is printed in response to the error number returned from the kernel. The kernel’s error reporting interface is quite limited and completely fixed. Adding text about `dmesg` is a shot in the dark by the error writer, trying to give the user more specific context to understand the error report.

The line about NFSv3 not being supported is the result of a collision between two attempts at helpfulness on the part of the programmer.

1. If an NFS mount fails, there is an automatic attempt to retry with a lower protocol version. This is useful, especially when servers are migrating to new protocol versions.
2. There is an attempt to catch obviously bad configurations early with sanity checks before too many computational resources are expended. These sanity checks provide an opportunity for more specific error reports.

The message comes from a sanity check that is intended to catch attempts to mount with a higher protocol version than the kernel is compiled with. The retry attempt (1) decrements the version counter, causing the sanity check to think 2 is the maximum allowed protocol version. At the same time, the retry attempt will still use the version specified on the command line if present (in this case the command line specified version 3). Mount, therefore, retries with v3 instead of v2, and the sanity check fails.

This can be fixed from within the program, but only to replace an incorrect message with a generic, useless one. Two isolated attempts at better error messages actually yield a worse error message: the whole can be less than the sum of its parts. This example shows some of the inherent limitations of error reporting from within a program.

2.2 End-user support

We plan to use Navel to build a distributed support service for non-commercial software. When something goes wrong that a user does not understand, the user can download the latest machine-learning models from a server machine (much like users download virus definitions for anti-malware software today). The model will determine the error based on the behavior profile. The behavior profile can be collected by reproducing the error, or leaving the Navel instrumentation “always on” (at some cost to performance that is quantified in Section 4.4). When the user presses the Navel button, Navel presents the user with a solution or work-around for their problem if the user community has found one. The following scenario illustrates how a complete software support system based on Navel will work.

1. A user experiences a software error that she or he cannot fix while running a Navel-instrumented program on their machine.
2. Program help menus do not provide sufficient help to the user.
3. The user activates the Navel monitor (probably already running as a system service) on the machine to read the behavior profile from the Navel-instrumented program and help diagnose the problem.
4. The Navel monitor classifies the problem, possibly consulting other machines on the Internet which contain repositories of trained machine-learning models and problem solutions.
5. The Navel system possibly returns a diagnosis (clear description of the error) and a digitally signed script that will fix it.

3. Representing program behavior

Navel builds a model of program behavior based on program measurement. The accuracy of the model, and the performance of Navel-instrumented programs is directly related to Navel’s behavior profiles. More detailed profiling will tend to require greater CPU and memory resources to collect, but will result in more accurate machine-learning models. There is a limit to this process however, because machine-learning models can be “overloaded” with too much information. There are many ways to map control flow information to a behavior profile, and this section explores different choices for that mapping, elucidating those decisions with some detailed examples.

The Navel prototype represents program behavior using information only related to control flow. Classification accuracy using only control flow information is high (see Section 4.3), and using control flow information minimizes the possibility of leaking sensitive user data.

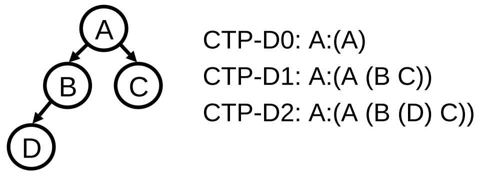

Figure 2. An example of call-tree profiling. The left side of the diagram is the function activation tree (the dynamic call graph) with arrows pointing in the direction of the function call. The right side are the subtrees for function A generated by the CTP algorithm for depth bounds from zero to two. Each subtree has parentheses indicating the depth of the function call in the tree.

The most successful machine-learning methods today (e.g., decision trees, support vector machines, boosting, etc.) classify based on a fixed-length vector representation of the data, called feature vectors by the machine-learning community. Navel builds behavior profiles that are fixed-length feature vectors, with the same number of features for every execution (though some features values might be zero for a particular execution). For instance, a feature vector could have an entry for every function, with the value of the entry is the number of times the function was executed.

3.1 Representation

This paper explores three approaches to behavior profiles that have different tradeoffs for performance overhead and level of execution detail. The first method uses *function counting* (sometimes called function call profiling [34]). Each function has a counter that is incremented when the function is executed. The order in which the functions are called is not retained. Function counting is efficient and tends to be accurate when each behavior has a set of unique functions associated with it.

Navel’s second behavior profiling method is *path profiling* as presented by Ball and Larus [3]. Each program path (unique sequence of basic blocks) has a counter that is incremented when the path is executed. Path profiling distinguishes well amongst program behaviors that result in different control flow within a function (intra-procedural control flow), something that function counting cannot do.

Navel’s third behavior profiling technique is new—*call-tree profiling* (CTP). Call-tree profiling associates a counter with a depth-bounded subtree of the program’s activation tree, and increments the counter when the subtree executes. The dynamic function calling behavior of a program can be represented by an activation tree, where each node is a dynamic instance of a function call, and edges are calls between functions. We use the notation *func:subtree* which means that the call to *subtree* originated in *func*. We use LISP style of tree representation for the subtrees where the left parenthesis indicates a call-depth increment and the right parenthesis indicates a call-depth decrement. Figure 2 shows an example of a small activation tree where function A calls function B which calls function D. These calls return and then function A calls function C. Activation trees grow large for non-trivial executions (about 43 million nodes for one execution of `gcc`).

Call-tree profiling captures some information about the order and relationship of function calls within a program—a rich source of program behavior data. In Figure 2 the depth one CTP (CTP-D1) feature value would be incremented whenever function A calls function B, and then function A calls function C. To limit the size of subtrees, Navel breaks subtrees at loop backedges. In Figure 2, the depth-one CTP feature is incremented only if A calls B and then A calls C within a single loop iteration within A. If A calls B in one loop iteration, then A calls C in the next loop iteration, that increments the value of two distinct CTP-D1 features :A:(A (B)) and A:((C)). Breaking subtrees at loop iteration boundaries reduces subtrees from hundreds of thousands of calls to less than a hundred. At depth-zero CTP is just function counting, because no calls made by a function will be counted. In this paper, CTP will refer to the union of the feature spaces at depth bound of at most two (i.e., CTP-D0, CTP-D1, and CTP-D2).

Like function counting, call-tree profiling only requires instrumentation at function entry points. Unlike function counting, call-tree profiling captures the dynamic calling behavior of functions, and more context. It captures some of the path information of path profiling (although it loses information about paths without function calls) and preserves some inter-procedural information.

Counts provide a robust profile of program execution, even if the program executes for a long time, and repeats the same actions. For example, if a server process executes the same recovery code five times and then writes a message to syslog, there is a robust 5:1 ratio of retry code to syslog writes. Navel uses sophisticated thresholding to make sure that rare events are not swamped by common events even though their counts might differ by orders of magnitude.

We do not consider behavior profiles of basic block counts because they would be expensive to collect, have little information gain over path profiling.

3.2 Example control-flow feature vectors

An example of Navel's program behavior profiles for the sample program shown in Figure 3 is shown in Table 1. All three behavior profiles are fixed length feature vectors that are presented to the machine-learning models as input. Each profile uses counters whose value is normalized by the total number of counted events in a run. Normalization allows comparison of runs with different input lengths, but must be done in a way that ensures rare events are not normalized to zero, so rare events are never lost.

Table 1 shows the feature vectors for each of the three profiles, for three runs of the sample program, each with different values for the variables n and c. Each row is a feature, and the column of normalized feature value counts for a given run comprises the feature vector for that run. Feature vectors for a particular profile (e.g., function counting) can be compared against each other, but not against vectors from another behavior profile. Note that both function counting and path profiling do not distinguish between $n = 0$, $c = 1$, and $n = 1$, $c = 1$ (the feature vectors are identical), while CTP-D2 does distinguish these cases.

For function counting, each function's normalized count is a feature. For path profiling, each path's normalized count is a feature. The paths are not explicit in the program listing, but the zeroth path in main and A correspond to the conditional being taken. For call-tree profiling, the normalized count for each depth-bounded subtree of the activation tree is a feature. The complete feature space is very large for path profiling and call-tree profiling, so Navel represents it sparsely, i.e., missing features are assumed zero valued.

4. Evaluation

To evaluate Navel, we built *Skepsis*, the Navel prototype. We evaluate Skepsis on large, mature programs that run on different operating systems (Linux and Windows). These programs exhibit a range of behavior that can confuse an end user, including ambiguous or unclear error messages, crashes, and, in one case, the exec of a shell due to a buffer overflow attack. Two benchmarks communicate with kernel modules.

```
void A(n) {
  if (n > 0) B();
}

void B() {}

int main() {
  int n = ... /*varies for each run*/
  int c = ... /*varies for each run*/
  if (c) A(n);
  B();
  if (c) A(1-n);
  return 0;
}
```

Figure 3. Sample C Program.

FC	n=0 c=1	n=1 c=1	n=0 c=0
main	0.2	0.2	0.5
A	0.4	0.4	0
B	0.4	0.4	0.5
PP	n=0, c=1	n=1, c=1	n=0,c=0
main_{p0}	0.2	0.2	0
main_{p1}	0	0	0.5
A_{p0}	0.2	0.2	0
A_{p1}	0.2	0.2	0
B_{p0}	0.4	0.4	0.5
CTP-D2	n=0, c=1	n=1, c=1	n=0, c=0
main:(main (A B A (B)))	0.2	0	0
main:(main (A (B) B A))	0	0.2	0
main:(main (B))	0	0	0.5
A:(A)	0.2	0.2	0
A:(A (B))	0.2	0.2	0
B:(B)	0.4	0.4	0.5

Table 1. Behavior profiles for three different executions of the sample program in Figure 3. **FC** stands for function counting; **PP** stands for path profiling; and **CTP-D2** stands for call-tree profiling with a depth bound of two.

4.1 Experimental setup

Each experiment executes an instrumented application with different inputs, and the application runs normally or generates a confusing or misleading error message (see Table 3 for the error message text). The accuracy we report is how well the machine-learning model is able to correctly classify the underlying error scenario for each confusing and ambiguous error message. A perfect classifier would correctly identify each error scenario from the behavior profile for each benchmark.

In order to train the machine-learning model there are roughly equal numbers of instances for each error class and the non-error class. This distribution is not intended to model the frequency of bugs occurring in the field, but rather trains the model to distinguish among the given cases without bias to any particular bug.

Skepsis uses static and dynamic binary translation to collect traces of runtime information. To experiment with different abstractions of program behavior, we collected basic block traces for all experiments and post-processed the trace differently for each behavior profile. For example, for function counting, each function entry basic block increments that function's counter.

For programs run on Linux we used the Pin [29] dynamic binary translation tool. For programs run on Windows we used the Phoenix compiler platform [32] to read and instrument Windows binaries. Skepsis runs as a system service, providing shared mem-

App	Reported error message	Improved error message	Recall	Prec	Comments
mpg321 0.2.10	Normal		68.9%	84.0%	Windows XP. `mpg321` is a command-line mp3 player run in batch mode to convert mp3 files to WAV format. Program ignores error reporting interfaces exported by important libraries.
	none	Audio frames frame data corrupted	87.3%	82.7%	
	none	ID3 tag data corrupted	95.8%	86.1%	
	none	Unsupported file format	98.7%	100.0%	
gzprintf zlib 1.1.3	Normal	none	88.0%	77.6%	Linux. A test harness randomly exercises the `gzprintf` function (which is known to have a buffer overflow), simulating normal program behavior before the crash or overflow. Code for all exploits were found on the Internet.
	none	Buffer overflow launches a shell	100.0%	100.0%	
	none	Buffer overflow exploit 1 causes crash	74.7%	86.2%	
	none	Buffer overflow exploit 2 causes crash	100.0%	100.0%	
gcc 3.1	Normal		100.0%	99.4%	Linux. The GNU C compiler which contains both hand-written and automatically generated source code, run on 4,070 pre-processed (".i" files) from Linux 2.6.13. A corrupter randomly modified working source code to create the errors.
	Syntax error on line where keyword, "else" appears	Syntax error on line where "`if(...) ;`" appears	99.7%	99.3%	
	Unexpected end of file	Missing close brace on a switch statement	95.7%	99.1%	
	Syntax error before X, where X varies	A semicolon is missing.	85.0%	88.7%	
	Syntax error before X, where X varies	Misspelled keyword	92.3%	86.7%	
FoxPro 9.0 alpha	Normal		100.0%	100.0%	Windows 2003 Server. Microsoft Visual FoxPro is a tool for creating database applications and components. We introduced the "hand induced" errors to simulate effects of memory corruption or extension hook misconfiguration.
	Access violation and quits	Attempted to save a Form over a file open in another program	100.0%	100.0%	
	Null pointer execution (hand induced)	Memory corruption	100.0%	100.0%	
	Jump to invalid memory (hand induced)	Extension hook misconfiguration or memory corruption	100.0%	100.0%	
latex teTeX 3.0	Normal		99.5%	97.3%	Linux. `latex` is a popular typesetting system. About 215 instances of each error class were evaluated. Instances were formed by corrupting a suite of varied latex files from multiple authors of academic papers. Latex prints the same error message for different underlying causes.
	! Extra alignment tab has been changed to \cr	Extra separator character (e.g &) in table, array or eqnarray	97.7%	100.0%	
	! Extra alignment tab has been changed to \cr	Reference to non-existent column in \cline command	100.0%	99.5%	
	! LaTeX Error: There's no line here to end.	Unexpected line break command in \item	98.1%	97.7%	
	! LaTeX Error: There's no line here to end.	Unexpected line break command inside center or flushleft or flushright environment	98.1%	96.8%	
	! Argument of \@sect has an extra }	Usage of fragile command \footnote inside \ section command	99.5%	100.0%	
	! Argument of \@sect has an extra }	Line break command used within \raggedright or \raggedleft environments	98.6%	98.6%	
	Missing number, treated as zero.	Missing numeric argument after \\	98.6%	99.1%	
	None	Warning: \\* will inhibit page break, not print an asterisk (which requires\\{*}).	97.7%	99.1%	
iptables 1.3.1	Normal		96.6%	96.3%	Linux. `iptables` is a widely used application in Linux for firewall configuration, network address translation (NAT), and packet filtering. It consists of user-level code and a kernel module communicating through the `netlink` [37] interface.
	iptables: Invalid argument	SNAT/DNAT/SAME rule applied to wrong chain	100.0%	100.0%	
	iptables: No chain/target/match by that name	Application of MARK to table other than mangle	100.0%	96.0%	
	iptables: No chain/target/match by that name	Missing kernel module	100.0%	100.0%	
	iptables: No chain/target/match by that name	Attempt to add rule to non-existent chain	96.3%	96.3 %	
iproute2 20041019	Normal		100.0%	97.9%	Linux. `iproute2`(this version from Debian Sarge distribution) is an advanced routing utility in Linux. It consists of user-level code and a kernel module communicating through the `netlink` [37] interface. The kernel returns `EEXIST` for the first two error classes, which is translated by the error reporting routine `perror` as, "File exists."
	File exists	User added colliding IP address	100.0%	100.0%	
	File exists	User added duplicate rule to routing table	100.0%	100.0%	
	None	User changed rule to conflict	98.3%	100.0%	
apache 1.3.36	Normal		100.0%	100.0%	Linux. `apache` is a widely used web-server application. One instance was collected per configuration. There are 6 normal classes: serving normal pages (HTML, image, directory), serving a missing file, and serving special status files `/server-info` and `/server-status`).
	No error message. `apachectl conftest` does not find this misconfiguration. `/server-info` returns 404 error	`mod_info` is missing	100.0%	100.0%	
	No error message. `apachectl conftest` does not find this misconfiguration. `/server-status` returns 404 error	`mod_server` is missing	100.0%	100.0%	
lynx 2.4.8	Normal		99.3%	100.0 %	Windows XP. `lynx` is a text based web browser. Error message consists of two redundant messages, each printed by a different code module, and the modules do not communicate.
	Alert! Unable to connect to remote host. Unable to locate remote host	Mistyped URL.	100.0%	100.0%	
	Alert! Unable to connect to remote host. Unable to locate remote host	Network Cable unplugged.	100.0 %	99.3%	
	Alert! Unable to connect to remote host. Unable to locate remote host	DNS Server not responding.	100.0%	100.0%	

Table 3. The **App** column has the application name and version number. The table then reports the error message given by the benchmark, and the improved error message that Navel would provide (or a brief description of the underlying error cause). Many error messages are ambiguous across multiple causes. **Recall** for each error is the number of correct instances with that error label divided by the total number of instances for that error class. **Prec**ision for each error class is the number of correctly classified instances of that error class divided by the total number of instances predicted as that error class (correctly or incorrectly). Recall and precision are two standard ways of thinking about prediction accuracy. The figures presented use the CTP behavior profile.

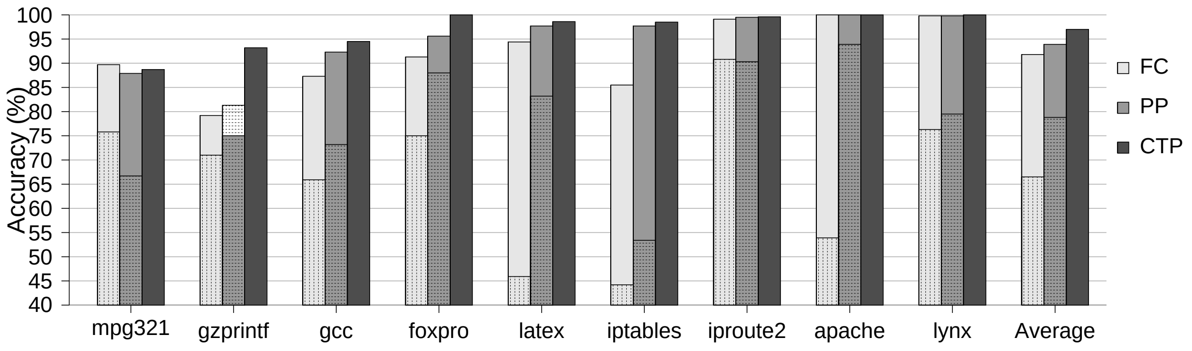

Figure 4. The figure shows the accuracy of the classifier used to distinguish the error cases, based on behavior profiles, for each benchmark. For each benchmark a classifier was built using three behavior profiles: function counting (FC), path profiling (PP), and call-tree profiling for depths zero, one, and two (CTP). The figure also presents sampled versions of function counting and path profiling with a sampling rate of 10%. The sampled accuracy is the stippled, lower bar in the stacked FC or PP entry.

App.	# of inst.	# of class	FC	PP	CTP
mpg321	282	4	202	25,450	1,229
gzprintf	600	4	104	1,489	396
gcc	1,582	5	2,293	40,513	50,623
FoxPro	184	4	4,724	17,738	63,383
latex	1,918	9	533	8,957	10,417
iptables	131	5	456	1,389	1,919
iproute2	146	4	146	392	475
apache	8192	8	605	1,611	3,052
lynx	615	4	696	3,483	4,540

Table 2. Navel model details for each benchmark. The second and third columns show the number of instances (program executions) and the number of error classes for each benchmark. The remaining columns show the number of features for each behavior representation.

ory buffers into which Skepsis-instrumented programs record their control flow. The use of shared memory ensures the record of program execution persists if the application abruptly terminates.

In order to generate a model that generalizes well, all experiments omit information from inside `libc`. Public entry points are included because it is application behavior when the application calls `strcpy`, but internal functions are omitted, because it is not application behavior when `malloc` calls an internal function, like `_int_malloc`. With a large enough class of traces, the machine-learning model will determine that `libc` internal functions are not relevant, but convergence of the model onto semantically meaningful features is faster if `libc` internals are omitted. The exception to this principle is the `gzprintf` benchmark, which is a thin wrapper around `libc` code.

Table 3 summarizes the benchmarks used in this study, including the confusing error messages they produce.

4.2 Machine learning model

Skepsis uses decision trees to build its model of application behavior. Decision trees are nested if-then-else statements and each leaf corresponds to a single class prediction. An advantage of decision trees (over more continuous methods like support vector machines (SVMs)) is their ease of interpretation. It is possible for a software engineer to validate the classifier based on knowledge of program structure. In the context of Navel, decision trees are as accurate as other machine-learning methods. Additional details about the machine-learning design of Navel are available [17].

Table 2 shows the number of features used to build the Navel model for each of the behavior profiles. More features means more information about control flow, which might lead to a more accurate model (if the model creation algorithm can deal with that many features). Because decision trees tend to be small relative to the number of features, they do not lose accuracy as the feature count increases, as is common in more continuous methods such as SVMs. The number of features for function counting is bounded by the total number of functions. Feature counts for path profiling are usually less than for call-tree profiling, though `mpg321` and `gzprintf` have more paths than call subtrees. Although the time to create the model increases with the number of features, the number of features does not affect the time required for the end user to run the model. For example, it takes 156 minutes to train the model for `gcc` when using CTP, but only takes 0.36 seconds to classify 158 execution traces.

4.3 Skepsis accuracy

Figure 4 shows the accuracy of Skepsis classifiers for a variety of benchmarks evaluated with different behavior profiles. These tables report accuracy using 10-fold cross validation, a standard technique for evaluating classifiers. The dataset is partitioned into ten sections, the classifier is trained and tested ten times; it is trained on nine sections of the data and its accuracy tested on the remaining tenth. The average of these ten tests is the reported accuracy of the classifier.

Function counting generally produces the least accurate classifiers, though its absolute accuracy value is surprisingly high. Path profiling is more accurate than function counting and call-tree profiling is more accurate than path profiling. Function counting for `mpg321` produces a model that is more accurate than call-tree profiling, but the difference is not statistically significant.

We present results for sampling function counting and path profiling, with a sampling rate of 10% (which is generous for systems that use sampling [27]). For example, the sampled function counts record one of every ten function calls, uniformly at random. The sampled results are the stippled part of each bar, achieving lower classification accuracy than non-sampled data in every case except path profiling for `gzprintf`. The poor performance of sampling confirms our intuition that sampling is the wrong approach for an error detection system. Navel must be sensitive to rare events.

If different errors exercise similar code paths, we would expect Navel's classification accuracy to degrade as the number of error classes increases. We trained classifiers on 2, 3,…,8 error classes for `latex`'s path profiling data and found classification accuracy degraded from 99.1% to 98.3%. One important area of future work is to test larger sets of error classes. Using the error text as a feature for the machine-learning model will help maintain accuracy while scaling to more error classes.

4.4 Performance

We evaluated Skepsis' performance while collecting behavior profiles for function counting. Navel modifies executables to keep an in-memory table of per-function counters. On Windows, we used the Microsoft Phoenix compiler infrastructure to modify binary programs, and on Linux we used Pin. We used a dual-processor Intel Xeon 3.0Ghz with 2GB of RAM running Microsoft Windows 2003 Server for `FoxPro`, a single-processor Intel Pentium 4 2.4Ghz with 1GB of RAM running Microsoft Windows XP for `mpg321`, and the same machine running the Linux ubuntu 5.10 distribution for `gcc` 3.1 and `latex`.

The `FoxPro` test used a benchmark available as part of the `FoxPro` package, which performs a variety of operations on a standardized set of tables. The workload for `mpg321` was conversion of 89 mp3 files to wav format - audio playback was disabled for the test. `gcc` was used to compile a 17,874 line ".i" file from Linux distribution, and `latex` was used to process an 1,175 line input file.

Phoenix function counting instrumentation slows down `mpg321` by 10% and `FoxPro` by 23%. Using Pin, function counting slows down `gcc` by 0.45% and `latex` by 1.0%. The Pin baseline numbers are run on Pin, the Phoenix baseline numbers use the pre-instrumented executable. Phoenix performance is lower than Pin because the binary instrumentation process deoptimizes the executable. Figures are the average of three runs.

Reviewer note: We are currently developing a fast online CTP algorithm. It is not correct enough to report in the paper, but our preliminary data shows a 10.6% slowdown on gcc, and a 10.9% slowdown on latex. We expect it to be more efficient than path profiling. It will be done for the final version of the paper.

The performance of path profiling is well documented in the literature. Ball and Larus [3] reported an average overhead of 44.8% for SPECINT95, and 31% for SPEC95 including floating point benchmarks. Their overhead varies depending on application, for example SPEC95's `gcc` overhead is 97%. Dynamic invariant detection techniques, such as DIDUCE [19], report a performance slowdown of $6\times$ to $20\times$. This slowdown prohibits using these techniques in customer-deployed applications. For future work, applying recent adaptive statistical profiling techniques [21] to path profiling could provide high quality control flow information to Navel at low cost.

5. Model analysis

This section examines the models built for the benchmarks evaluated in the previous section to validate that they are built on semantically meaningful program features. It is desirable for a model to use meaningful program features because that gives confidence that the model will generalize well, and it suggests that a programmer can look at the Navel model and get useful debugging information.

This section shows that decision trees produced by Navel often reveal the complexity of processing along error paths. Intuition might suggest that there is a simple bijection between error behavior and functions with names like `error`, but we have found that applications are not consistent in this respect—many errors are reported via the same routines. Moreover, some applications use their error reporting routines to report warnings when there is no error.

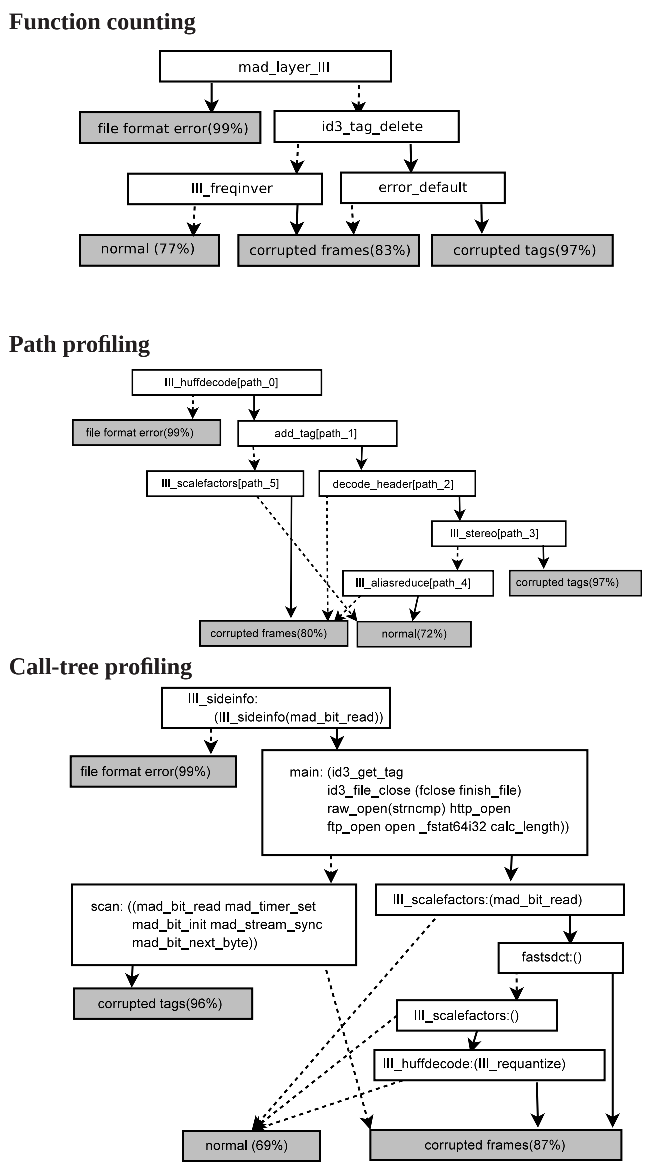

Figure 5. Decision trees produced for the `mpg321` benchmark. Dotted lines are taken when the normalized count of the feature value is less than or equal to a threshold, while the solid line is taken when it is greater than the threshold. The threshold is determined automatically for each benchmark by the decision tree algorithm, and can be different for each node in the tree. Clear boxes are features. Function counting features are normalized function counts, path profiling features are normalized path counts (identifiers in brackets are path identifiers), and call-tree profiling features are normalized counts of call subtrees (represented by the symbolic tree names in brackets, with function names for nodes in each call tree). Shaded boxes are error classes.

5.1 mpg321 model example

Figure 5 shows the decision tree models created by the Navel function counting, path profiling, and call-tree profiling behavior profiles for the mp3 player `mpg321`. The different trees show how each behavior profile provides different clues to the machine-learning model about the same underlying behavior. The same underlying program behavior is reflected into different behavior profiles, and the decision tree based on the different profiles provides more or less classification accuracy.

In the function counting tree we see the simplest set of rules that depict differences in control flow across the four error classes. At the root of the tree, the function mad_layer_III provides near perfect discriminative information for the wav error class: the mad_layer_III routine is part of the libmad library and is called when the audio frame decoder runs. Since the wav format is among the formats not supported by mpg321, it will not successfully decode any audio frames, and the libmad library will never call mad_layer_III. The id3_tag_delete routine differentiates between the corrupted tag and and other classes. The ID3 tag parser in the libid3tag library dynamically allocates memory to represent tags and frees them with id3_tag_delete. If tag parsing fails, the memory for a tag is not allocated. Since no tag parsing succeeds in the corrupted frames case, id3_tag_delete is never called to free the tag memory, making it's absence discriminative for that class. The libmad audio library's default error handler error_default is used if the application does not specify one. mpg321 does not specify its own error handler, so the presence of the function indicates corrupted audio frames, and its absence indicates the corrupted id3 tags case. Finally, III_freqinver, which performs subband frequency inversion for odd sample lines, is called very frequently as part of the normal process of decoding audio frame data. When there are corrupted frames, this function is called less frequently, and the decision tree algorithm finds an appropriate threshold value to separate the normal from the corrupted case.

The path-profiling tree latches onto similar behavioral features of the execution as function-counting tree does, but because it has access to intra-procedural control flow, it chooses different features. For instance, it uses a path through III_huffdecode to distinguish an unsupported file type. III_huffdecode will be called whether the input file is an mp3, or an unsupported wav file, however; the path through III_huffdecode will reflect the error if the input format is wav, making it a perfect indicator for that error class. The add_tag function is called when the libid3tag library tag parser succeeds. Absence of this call provides a good heuristic for detecting the corrupted frames class. Finally, III_scalefactors is called for every decoded audio frame. We expect fewer successful frame decodes for the corrupted frames class, making a threshold on a successful path through III_scalefactors a good differentiator between the normal and corrupted frames classes.

The decision tree built on call-tree profiling data has the richest combination of data sources of any of the decision trees because call-tree profiling provides the most data about different execution scenarios. Call-tree profiling uses the presence of the libmad library function III_sideinfo (which decodes frame side information from a bitstream) calling the utility function mad_bit_read as an indicator of successful audio frame decoding. The lack of that calling pattern reliably indicates a file format error. The corrupted frames class is once again differentiated from the normal class by a threshold value on a subtree of libmad functions that will only be called during successful decoding of audio frame data, such as III_scalefactors, the discrete cosine transform function fastsdct, III_huffdecode, and so on. The libmad function scan encapsulates the process of reading mp3 files. A CTP rule wherein scan calls a function that calls a number of low-level stream manipulation routines such as mad_bit_read, and mad_timer_set, and so on, provides discriminative power in combination with a similarly complex control flow pattern in main for the corrupted tags error class. The decision tree node whose CTP rule involves main, id3_get_tag, and so on differentiates between normal and error conditions for the handling of ID3 tags, while the decision tree node whose CTP rule involves scan discriminates between successful and unsuccessful audio decoding. The high level pattern exposed by these rules is the combination of failed ID3 tag parsing with successful audio decoding, which precisely describes the corrupted tag error class.

```
  ...
  if (THEN_CLAUSE (t))
    expand_stmt (THEN_CLAUSE (t));
  if (ELSE_CLAUSE (t))
    {
      expand_start_else ();
      expand_stmt (ELSE_CLAUSE (t));
    }
  expand_end_cond ();
}
```

Figure 6. gcc source code for function genrtl_if_stmt from c-semantics.c, line 397.

5.2 Gcc and the difficulty of good error reporting

We present an example from gcc, the GNU C compiler, where adding good error reporting would complicate the source code unacceptably.

```
if (test);      // extra semi-colon here
{ /* do something */ }
else
{ /* do something else */ }
```

The programmer has accidentally typed an extra semicolon at the end of an if clause. gcc's error message is "*parse error before "else"*", and it identifies the line on which the else keyword is located (which could be many lines away from the real problem). The compiler provides no information about the actual cause of the problem—the extra semicolon. Seasoned programmers train themselves to ignore the line numbers of certain classes of errors, but this is no justification for the compiler's behavior. We should expect more from software.

Figure 6 shows source code for gcc 3.1 that handles parsing if statements.[1] When the compiler encounters the semicolon, the two tests in genrtl_if_stmt evaluate to false. While it might be tempting to print a meaningful error message in this situation, it would be incorrect. This is because the C language specification allows an if statement with no corresponding body or else clause. The problem only exists, and is only detected, when reading the else keyword. Trying to search backward from the else to the problematic if (or forward from the if to the else) requires additional data structures and requires complicating genrtl_if_stmt to update those data structures.

The Navel decision tree can correlate the path through genrtl_if_stmt where each test fails with the path that reports the else parse error to disambiguate a particular underlying cause for this error. The error case can be distinguished without adding code to gcc. Navel enables greater functionality for applications without changing them.

For the program error of missing closing bracket for a switch statement, path profiling detects paths through the automatically-generated function yyparse_1 due to receiving an end of file

[1] In an attempt to produce as many useful error messages as possible on a single invocation, gcc calls into its semantic processing routines even if parsing fails.

token in an unexpected state. Function counting, using coarser-grained information, distinguishes the non-error case as one that does not call the diagnostic formatting function `context_as_prefix`.

5.3 Analysis of remaining benchmarks

latex. The `showcontext` function is generic and displays error messages. In the case of function counting this acts as the discriminative feature for non-error instances since it is not called during normal execution. Function counting uses the frequency of calls to `showcontext` to distinguish error cases. However, path profiling extracts much more information by looking at paths within `showcontext` to discriminate between different error classes.

Similarly, `finalign` is called in both error and non-error cases that use tables, but the execution path within the function distinguishes the error case from normal execution. One of our error class consisted of supplying a non-numeric argument to the line break command when it was anticipating a numeric value. The machine-learning model successfully pinpoints the error source to the `scanint` function, which is called expecting a numeric argument.

One interesting `latex` example is where latex support web sites offer more information than the `latex` error message itself, but do not include all possible causes for the error. If a table, array or `eqnarray` has more separator characters (ampersands) than columns, `latex` prints the obscure error message, "*!Extra alignment tab has been changed to* `\cr`". Most `latex` books and most `latex` support websites recommend checking the number of ampersands if a user receives this error. Some websites and books are helpful enough to suggest a missing end of row symbol (\\\\) on the previous line. However, the error message is not unique: misuse of the `\cline` command, a directive that draws a horizontal line in the table will result in the same message if one of the arguments to `\cline` refers to a non-existent column in the table.

Navel connects a user to the solution for their particular problem, not just the most popular cause of a given problem. Most error messages are intended to apply only to the most likely scenario that can cause it.

gzprintf. It is a strength of Navel that a particular library or function can be examined in isolation from the rest of an application. Because Skepsis maps control flow in a composable way, small bits of code can be examined carefully.

apache. Apache is a widely used webserver. It has many configuration parameters, complicating the task of correct configuration. Apache error logging and configuration syntax checking tools are helpful, but many important functions of Apache are implemented in dynamically loaded modules. When apache encounters a configuration error, it calls each module to check for an appropriate error handler. If no loaded module is able to handle the error, apache ignores the error or reports it as a misspelled keyword.

When `mod_status` or `mod_info` is loaded, "SetHandler" binds to a specific URL to get server status via HTTP. If modules are missing, even though the SetHandler was configured, apache returns a 403 error, and administrators get no hint that the error arises from misconfiguration. Skepsis is able to classify these behaviors with 100% accuracy.

6. Reducing human effort

Navel leverages the expertise of a few software users for the benefit of the entire user community, but further reducing human effort makes Navel more attractive. This section discusses two methods to reduce human effort—unsupervised clustering of error behavior and using a model trained for one version of software on the next version.

Benchmark	FC Cluster	FC Model
mpg321	79%	72%
gzprintf	66%	75%
gcc	59%	87%
FoxPro	98%	91%
latex	25%	94%
iptables	54%	86%
iproute2	N/A	99%
apache	78%	100%
lynx	64%	100%

Table 4. Accuracy of error labels using unsupervised clustering, and a machine-learning model trained using labeled instances. Benchmarks use the function counting behavior profile.

6.1 Clustering errors

Manually determining whether an error execution is an instance of a known problem is laborious. When possible, Navel should cluster behavior profiles of error executions without human involvement. If a single profile in a cluster is labeled, Navel can speculatively use that label for the other profiles in the cluster.

At the core of any clustering algorithm is a function that quantifies the similarity between two instances, and the clustering objective is to find a set of clusters with high intra-cluster similarity. For example, the popular k-means algorithm uses squared Euclidean distance, which is computed as the sum of the squares of the distances between each feature value. For Navel, feature values take on a wide range of values; some functions are called tens of thousands of times in a program instance, while other functions are called only 5 or 10 times. Consequently, for most clustering algorithms, the underlying similarity scores computed will reflect the similarity between only a handful of highly called functions (or paths or call subtrees).

One approach to alleviate this problem is to represent each feature value in binary 0/1 form. This can be done by giving features with counts of zero a new value of zero and giving features with counts larger than zero a new value of one. Although this method yields significant improvement in the clustering results, Navel uses a more general binarization approach by thresholding at an arbitrary value because features often take on only a few distinct, non-zero values. Navel finds the split that (1) minimizes the entropy (i.e. maximize the homogeneity) of the sets induced by the split and (2) also maximizes the quality of the split by preferring 50/50 splits. The exact criteria Navel uses is a standard information theoretic measure called mutual information [15], and quantifies the similarity between two random variables (in this case, the threshold function and the values the feature takes on). Finally, Navel uses a variant of the k-means algorithm, spherical k-means [18], which has been shown to work well in other domains with large numbers of features, such as text clustering.

Table 4 compares the accuracy of behavior profiles labeled using unsupervised clustering and classified using a model trained with labeled profiles. The quality of clustering is erratic, with `FoxPro` clustering more accurate than the trained model, while the `latex` clusters distinguish different kinds of errors, but do no better than random at disambiguating causes of a single error report.

In general, clustering does significantly better than random guessing. Clustering allows Navel to match users with a problem workaround even before other users have labeled profile of that problem. With no human involvement, a model trained with clustered labels could be useful to a developer for understanding a particular error behavior.

Clustering is useful to allow two users to determine if they have instances of the same error, or if a user has an instance of a popular

Profile	3.1	3.1 train, 3.2 use	3.2
FC	87.3% ± 1.5%	85.9% ± 1.6%	84.8% ± 1.7%
CTP-D1	91.5% ± 1.3%	90.3% ± 1.4%	91.3% ± 1.3%
CTP-D2	92.9% ± 1.2%	93.5% ± 1.2%	92.6% ± 1.2%

Table 5. Accuracy, along with 95% confidence intervals, for training a classifier on `gcc` version 3.1 and using it to classify errors for version 3.2. For comparison accuracy results for training classifiers for version 3.1 and version 3.2 are given.

error. Developers can use clustering to prioritize investigating errors that are popular in field deployments. When developers receive error reports, they would like to know if the errors are distributed evenly over all of the erroneous classes, or are there "hot" errors cases that many users are encountering. With this information, developers can prioritize the hot errors over less common ones.

6.2 Classifying without retraining

Software release schedules are tight, and it would be convenient if developers did not have to completely retrain a Navel classifier for a minor release. Navel can generate a classifier for a new application version by eliminating features that refer to functions or paths that do not exist or were changed in the new version of the code. Because most code does not change between versions, Navel does not need to construct a complicated map between versions to maintain an accurate model.

Table 5 shows the results of this strategy using Skepsis. Since most code does not change between versions, the features chosen as discriminative for `gcc` 3.1 are still discriminative for version 3.2. There are enough features in the Navel model that the absence of a particular feature usually does not significantly reduce the accuracy of the model. The mean accuracy of the classifier for version 3.1 and used on 3.2 is greater than the accuracy of the classifier trained on 3.2, but the difference is within the 95% confidence interval.

7. Related work

Integrating machine learning with program understanding is an active area of current research. We summarize the major research threads below, but first position Navel among them. Navel uses data about program structure (control flow), as opposed to dynamic invariants [10, 19]. Navel characterizes any kind of program behavior, not just crashes or program errors [10, 19, 27, 34]. Navel works within the scope of a single program, not a distributed system [14, 5, 12, 1]. Navel records information throughout a program's history, not just information available at crash time, like a stack backtrace [30, 31, 7]. There are some tools that automate some end-user support tasks, for instance performance diagnosis systems [5, 14], and configuration debugging systems [40, 24], but these systems do not have the fine-grained resolution of Navel.

A group at Microsoft Research has identified the same benefits that we have from correlating low-level system events with error reports to automate problem diagnosis [43, 44], because the problem is important and practical. They currently focus on building models from sequences of system calls instead of compiler/runtime system instrumentation. Our methods for feature selection and model training differ, and it would be interesting to directly compare the techniques. Our paper includes novel methods to reduce human effort in the system.

IBM has a system to classify stack backtraces harvested on a crash [9], and the technology has been deployed in tier TrapFinder tool. Their motivation is similar to Navel's—reduce the human effort needed to match problems from different program executions. Navel diagnoses a wider range of problems than crashes, and it operates on behavior profiles, which are a richer source of data than stack backtraces.

Statistical bug isolation [27, 26] also correlates low-level application behavior with application behavior (bugs) and builds a model. The systems differ in their source of program data. Statistical bug isolation samples program invariants rather than gathering information about program control flow. Section 4.3 demonstrates a sharp loss of accuracy if Navel uses sampling. Statistical bug isolation must eliminate sub-bug and super-bug predictors; Navel has an analogous struggle to gain enough training instances to isolate the program behavior created by the error condition. Navel is intended to provide support to the user, while statistical bug isolation is intended for developers; the systems are complementary.

Podgurski *et al.* [34] identify a similar motivation to Navel and they also investigate `gcc` behavior. This paper presents more alternatives for sources of program information, presents data on more programs and problem classes, presents higher classification accuracy, and introduces extensions to apply learned models to libraries and different versions of the same program.

Bowring *et al.* [8] models software behavior as Markov models using control flow between basic blocks and then uses active learning to cluster the models. Navel benchmarks are orders of magnitude larger (e.g., 8,654 functions for `gcc` and 6,363 for `FoxPro` versus 136 functions for SPACE).

Liu *et al.* [28] use program behavior graphs as features for a machine-learning model just as Navel uses data related to program control flow. The number of program behavior graphs grows quickly with program size, and can become computationally intractable even for the small Siemens programs [23].

DIDUCE [19] uses dynamic program invariants to detect program behavioral anomalies. The anomalies can indicate program bugs, but at a performance slowdown of 6–20×. Navel can classify program behavior that is not anomalous. PeerPressure [40] identifies anomalous configurations in the Windows registry using Bayesian statistics, and does not deal with program behavior.

Navel classifies executions of a single program, while many published systems classify behavior in a distributed system [14, 5, 12, 1]. The performance constraints in a single program are more stringent. Most of these systems look at interactions between software components, and only attempt coarse-grained identification of a faulty or slow component. Navel currently uses functions or paths as a basic building block of behavior, but it might be able to leverage some of these published algorithms by monitoring interactions between libraries or even object files within the program.

Microsoft's Dr. Watson tool [30] and Gnome's BugBuddy [7] can collect information from the end-user to find where the application crashed. These tools are limited to collecting information at the time of the crash, usually a list of loaded modules, a stack backtrace and some stack memory contents.

SimPoint [39] characterizes the phase behavior of applications using basic block execution counts to maintain the accuracy of architectural simulation while executing fewer instructions. The types of program behavior it detects occur over much longer time windows than the errors that Navel detects.

Program paths [3] have been used to analyze runtime program behavior. Path profiling techniques [3, 33] pinpoint the "hot paths" which are frequently executed, as well as those with variable execution times. They are effective for optimizing program performance, as well as identifying areas requiring more test coverage. Path Spectra [36] approximate an execution's behavior with the occurrence (or frequency) of the individual paths. Spectral differences have been used to identify the portions of a program's execution that differ with different inputs, notably, during Y2K testing [20]. Path Spectra focused on identifying path differences between several program runs, whereas Navel's novel use of path profiling uses machine learning to identify which paths are *common* to each error class. Whole program paths [25] is a more recent technique that uses a lossless compression of the program's control flow, which is able to capture loop and inter-procedural information. Although Navel's call-tree profiling and whole program paths have similar aims—obtaining a compact abstraction of program control flow while preserving time-series information, the two differ in two key aspects. First, CTP operates at a function-level granularity, instead of individual paths (and thus may permit lower overhead profiling). Second and more importantly, CTP's compression results in rules that are directly comparable from one program execution to the next. Whole program paths use a compression scheme to reduce overhead, with no concern for whether the grammar rules that allow the compression change from run to run—a key requirement for Navel.

Context sensitive profiling [2] uses calling context trees that contain more calling context than a call graph, but are much smaller than the program's full activation tree (dynamic call tree). Calling context trees ignore loops, and so are not directly useful for Navel. They also include less ordering information than CTP, but might be useful as an alternate behavior profiling technique.

Several systems have been proposed that automatically recover from software problems. Chronus [41] uses user-provided software probes to search through previous system states to find the point at which the system first failed due to misconfiguration. The Rx [35] system relies on a set of *sensors* (e.g., OS signals) to detect that a program is behaving abnormally. To recover, Rx rolls back the application state to one which was captured by a previous checkpoint, and changes the application's environment before allowing it to proceed. Both Chronus and Rx attempt to detect failures and correct them, while minimizing the user-visible effects. Navel addresses error messages that do not represent a failure of the program, but the messages do not provide enough information to the user to fix their problem. An ideal system would combine the two approaches by first attempting to automatically correct errors and then using a Navel-like service when an application must report an error to the user.

8. Future work

Navel creates machine-learning models from program behavior. This section discusses ways to extend Navel's reach into the operating system. It also discusses Navel's use for management of bug databases.

8.1 Behavior profile of the operating system

Some error cases cannot be classified with only user-space behavior profiles. For instance, when an application fails to look up a hostname in DNS, it cannot distinguish between an incorrect DNS server name and an unplugged network cable without further information from the operating system. Other classes of problems requiring OS behavior profiles include problems that arise from the interaction of more than one process and applications that are split between user and kernel space, such as `iptables` which consists of a user-mode program and a kernel module. One advantage of getting a behavior profile for the OS is that monitoring certain classes of OS events, such as system calls, is less resource intensive than monitoring application control flow.

8.2 Bug databases

Duplicate entries in bug databases are common. As an example, one in six resolved bugs in the `gcc` bug database was closed by finding that it is a duplicate of a previous entry (3,288 out of 20,459). The twenty most reported issues accounted for 434 entries, and duplication is common even among less popular bugs. Duplicate bugs waste the time of developers who are assigned to investigating filed issues, as well as the time of users who are trying to search the database.

If bug descriptions were augmented with behavior profiles and Navel models, duplicate entries could be detected without humans reasoning about the text of error descriptions. When a new bug report is entered, the system can check to see if there is a bug with a similar behavior profile, whether or not the human who entered the bug used the same words to describe it. Administrators can review the database for entries with similar profiles that might be duplicates.

Searching bug databases with a behavior profile has advantages over keyword searches because the Navel system will generate the behavior profile for the user, and the profile does not suffer from the imprecision of language. Searching for "outlook crash" returns 8.7 million hits on Google, which indicates how a simple textual description of a computer problem can be highly ambiguous.

9. Conclusion

We present Navel, a system that can improve error reporting by classifying behavior profiles. Navel monitors program execution, creating the behavior profile which is a summary of the program's behavior. It trains a machine learning model to discriminate between different behavior profiles. Classification with the model allows an average user easily to determine if the problem they just experienced is one for which the vendor or support organization has a fix or workaround. Our prototype implementation, Skepsis, accurately and efficiently classifies behavior of large, mature applications that demonstrate baffling error behavior.

10. Acknowledgements

This work was funded in part by a grant from the Phoenix group at Microsoft Corporation.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, Bolton Landing, NY, Oct. 2003.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performacne counters with flow and context sensitive profiling. In *PLDI '97*, pages 4–16, June 1997.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, 1996.

[4] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, 2002.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[6] R. Barrett, E. Haber, E. Kandogan, P. P. Maglio, M. Prabaker, and L. A. Takayama. Field studies of computer system administrators: Analysis of system management tools and practices. In *ACM CSCW (Computer-supported Cooperative Work)*, 2004.

[7] J. Berkman. *Bug-buddy — GNOME bug-reporting utility*, 2004. `http://directory.fsf.org/All_Packages_in_Directory/bugbuddy.html`.

[8] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA*, Jul 2004.

[9] M. Brodie, Sheng Ma, G. Lohman, L. Mignet, N. Modani, M. Wilding, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Second International Conference on Autonomic Computing, 2005.*, pages 101–110, 2005.

[10] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, 2004.

[11] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[12] M. Y. Chen, E. Kiciman, E. Fratkin, A.o Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[13] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[14] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP*, 2005.

[15] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley Series in Telecommunications, 1991.

[16] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[17] Jason V. Davis, Jungwoo Ha, Christopher J. Rossbach, Hany Ramadan, Indrajit Roy, and Emmett Witchel. Autonomically improving software error messages with machine learning. In *Submitted to AAAI*, 2006.

[18] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., New York, 2001.

[19] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[20] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. In *Journal of Software Testing, Verification and Reliability, vol 10, no 3*, 2000.

[21] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI*, 2004.

[22] J. Humphreys and V. Turner. On-demand enterprises and utility computing: A current market assessment and outlook. Technical report, IDC, Jul 2004.

[23] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE*, 1994.

[24] N. Lao, J. Wen, W. Ma, and Y. Wang. Combining high level symptom descriptions and low level state information for configuration fault diagnosis. In *LISA*, 2004.

[25] J. R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269, 1999.

[26] B. Liblit, A. Aiken, A.X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[27] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[28] C. Liu, X. Yang, H.Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proc. of 2005 SIAM Int. Conf. on Data Mining (SDM05)*, 2005.

[29] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[30] Microsoft Corporation. *Dr. Watson Overview*, 2002. `http://www.microsoft.com/TechNet/prodtechnol/winxppro/proddocs/drwatson_overview.asp`.

[31] Microsoft Corporation. *Online Crash Analysis*, 2004. `http://oca.microsoft.com/`.

[32] Microsoft Corporation. *Phoenix Compiler*, 2005. `http://research.microsoft.com/phoenix/`.

[33] E. Perelman, T. Chilimbi, and B. Calder. Variational path profiling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[34] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE*, 2003.

[35] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies – a safe method to survive software failures. In *ACM Symposium on Operating System Principles*, 2005.

[36] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449. Springer–Verlag, 1997.

[37] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetov. Linux netlink as an ip services protocol. RFC 3549, 2003.

[38] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, 2001.

[39] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, Oct 2002.

[40] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[41] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Diagnosing computer problems using time travel. In *11th SIGOPS European Workshop*, 2004.

[42] Y. Xie and D. Engler. Using redundancies to find errors. In *SIGSOFT '02/FSE-10*, 2002.

[43] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma. Automated known problem diagnosis with event traces. *MSR-TR-2005-81*, 2005.

[44] C. Yuan, N. Lao, J. Wen, J. Li, Z. Zhang, Y. Wang, and W. Ma. Automated known problem diagnosis with event traces. In *EuroSys*, 2006.