Using causality to diagnose configuration bugs

Mona Attariyan and Jason Flinn
Computer Science and Engineering
University of Michigan

Abstract

We present a novel method for diagnosing configuration management errors. Our proposed approach deduces the state of a buggy computer by running predicates that test system correctness and comparing the resulting execution to that generated by running the same predicates on a reference computer. Our approach generates signatures that represent the execution path of a predicate by recording the causal dependencies of its execution. Our results show that comparisons based on dependency sets significantly outperform comparisons based on predicate success or failure, uniquely identifying the correct bug 86–100% of the time. In the remaining cases, the dependency set method identifies the correct bug as one of two equally likely bugs.

1 Introduction

Software in modern computer systems is extraordinarily complex. Many applications have a large number of configuration options that can customize their behavior. Further, each application interacts with the other software on a computer through channels such as shared libraries, registry entries, environment variables and shared configuration files. This flexibility has a cost: when something goes wrong, fixing a configuration problem can be both time-consuming and frustrating. Consequently, there has been considerable effort by the research community to simplify configuration management [2, 5, 8, 9, 10, 11].

The process of configuration management can be divided into two separate tasks: diagnosing which specific problem is afflicting the computer system, and determining how to fix that problem. In this paper, we address the former task: finding the root cause of a configuration problem. We assume that the bug is known, i.e., the problem has been previously encountered and solved on a *reference computer*. The reference computer could be a test system used by software developers or a personal computer owned by a peer who had the same problem. Thus, the problem of diagnosing an unknown bug on a *sick computer* can be reduced to identifying that the sick computer is in a state similar to a buggy state on the reference computer for which a solution is known.

To deduce similarity between states on the reference and sick computers, our approach is to run a set of *predicates* that test the correctness of the computer system. In previous work [8], we used the success or failure of predicates to deduce similarity. While this approach is intuitive, we have since encountered several drawbacks. First, an expert, e.g., a software developer or tester, must craft a predicate to cover each new bug. Second, a single predicate may often detect many bugs, causing many states to appear similar. Finally, a test case that is too finely crafted to the reference computer may inadvertently report an error due to a benign difference between the environments of the sick and reference computers.

We present a method for diagnosing bugs that uses signatures derived from the set of objects upon which each predicate's execution causally depends. We use system call tracing tools such as `strace` to record each predicate's *dependency set*, i.e., the files, devices, fifos, etc. read by the predicate. We compare the dependency sets generated on the reference and sick computers to deduce similarity. Our results show that comparisons based on dependency sets significantly outperform comparisons based on predicate success or failure, uniquely identifying the correct bug 86–100% of the time. In the remaining cases, the dependency set method identifies the correct bug as one of two equally likely bugs.

2 Background

Our previous work in configuration management, titled AutoBash [8], used the pattern of success and failure of known predicates to diagnose configuration errors. Using this approach, AutoBash executes all predicates, $\{P_0, P_1, ..., P_n\}$ on the sick machine and aggregates their results as a binary vector $S_{current} = \{1, 0, ..., 1\}$ (with 1 indicating success and 0 failure). AutoBash then compares $S_{current}$ with a set of system state vectors S_i from $\{S_0, S_1, ..., S_m\}$, where each system state was generated by running the predicates on the reference computer prior to fixing a known bug. Intuitively, each vector is a signature for a system state that represents a particular bug. Thus, AutoBash chooses the system state vector that is most similar to $S_{current}$ as the most likely diagnosis for the bug. According to the diagnosis, AutoBash chooses a solution from its database and speculatively runs the solution. Then, AutoBash tests the affected predicates to determine whether the problem is fixed or not. If the problem is fixed, AutoBash commits the solution; otherwise, the solution is rolled back and AutoBash tries the next most likely diagnosis. The accuracy of diagnosis determines how fast AutoBash can find a correct solution. As the AutoBash diagnosis method uses the Hamming distance as a similarity metric, we will refer to it as the *Hamming distance method*.

One advantage of the Hamming distance method is that it treats predicates as black boxes. AutoBash does not need to understand what each predicate does; it only needs to execute each predicate as a child process and check the return code to determine success or failure. Another advantage is portability; since predicates are application-level test cases, their success or failure should not be perturbed by irrelevant fluctuations in the application environment such as variations in the operating system or installed software.

However, as Section 5 shows, the Hamming distance method suffers from ambiguity. Since the similarity metric takes into account only the success or failure of predicates, many different bugs may have identical state vectors. To allow correct diagnoses, a tester or developer must painstakingly craft specific predicates that target each known bug. Easy-to-create stress tests, which we refer to as *kitchen sink predicates*, are useless because they fail for most bugs. For example, a Linux kernel compile can trigger many possible compiler configuration bugs, so its failure tells little about the underlying system state. On the other hand, failure of a hand-crafted predicate that only checks a specific kernel header reveals much more about the bug. However, writing such predicates to cover all known bugs takes a lot of effort.

Another drawback of the Hamming distance method is lack of granularity: many system state vectors may lie at a Hamming distance of one or two from a given result vector, even though each state causes a different set of predicates to fail.

3 Design

Based on our observations, we tried to design a method that would retain the advantages of the Hamming distance method while eliminating its disadvantages.

Looking more closely, we realized that although the success or failure of predicates may be similar for many bugs, the execution paths of those predicates usually differ for each bug. For example, if a predicate compiles and runs a program, any bug in the compilation, linking or loading phases can cause the predicate to fail. However, bugs in each of the three phases cause the predicate to take different execution paths. As another example, a configure script takes different execution paths depending upon the particular software that is installed on a computer. Thus, if we can generate a signature that captures the execution path of a predicate, we should be able to more precisely identify a configuration error.

Ideally, we would like to generate a signature that is precise enough to capture different execution paths that are induced by different configuration bugs. However, the signature should be robust enough so that executing a predicate on computers with the same bug but different operating systems, installed software, and execution environments generates similar signatures. For example, we could use all the system calls executed by a program to generate a signature for the execution path [4, 12]. However, random permutations caused by thread scheduling, interactions with other processes, and other sources of non-determinism will cause the sequence of system calls to vary even when a predicate is executed on the same platform. Further, this method would perform poorly for our purposes because we run the same predicate on two computers with different software. For example, the sequence of system calls will change with different versions of shared libraries such as libc, with different versions of the same operating system, or with different operating systems.

To generate a more robust signature, we decided to instead use the causal dependencies of predicate execution as a signature. We define the dependency set of a process to be the set of files, directory entries, file metadata, devices, fifos, and other objects read by the process and its descendants during their execution. This choice is based on the observation that the layout of application files and directories shows only minor fluctuations across platforms. Further, the concept of files and directories is common to most operating systems, while specific system calls differ greatly. At the same time, the dependency set usually reflects significant differences in the execution paths of a predicate in the presence of different bugs. For instance, in the above compilation example, if the predicate fails in compilation, the predicate's dependency set will not contain any objects related to the linker or loader simply because execution ended before those phases. Therefore, the dependency set can capture the progress of predicate execution and generate different signatures for different failures.

There are several possible approaches for generating dependency sets. We wished to avoid intrusive monitoring methods that require the application under test or the host operating system to be modified. We also wanted to reuse existing tools as much as possible. We observed that most operating systems have a system call tracing tool such as Linux's `strace` or FreeBSD's `ktrace`. We wrote parsing programs that take tracing tool output and generate the corresponding dependency set. The only drawback of these tools is that they can only trace the main process and its descendants. Activities of other processes communicating with the main process and its descendants via shared memory, pipes or files cannot be automatically traced with these tools. To address this issue, we could trace all processes in the system. However, we judged that tracing all processes would incur a lot of overhead while adding negligible accuracy.

4 Implementation

We use strace and ktrace to generate dependency sets on Linux and FreeBSD, respectively. These tools intercept all system calls made by a process and its descendants along with their parameters and return values. We trace each predicate and pipe the tool output to a parser that calculates the predicate's dependency set.

The parser divides system calls into three categories. The first category consists of system calls that do not affect the dependency set of the predicate. For example, the brk, mmap and mprotect system calls manage a process's memory. The parser simply ignores these system calls. The second category consists of system calls that do not directly affect the dependency set but may change the objects that are added later. For example, the fchdir system call changes the current directory to the file descriptor specified by its first parameter. This system call does not change the dependency set, but it affects all following file names with relative paths.

The third category consists of system calls that directly affect the dependency set. For each system call, the parser adds appropriate dependency records to the process's dependency set. For example, the stat system call provides information about a specified file. A successful stat system call makes the process dependent on the directory entry and metadata of the specified file, as well as the directory entries and metadata of all directories in the file path. As another example, reading from a file makes a process dependent on the content of the specified file, as well as its metadata.

Before processing the parameters of a system call, we check the return value and error type. Without considering the return value, we are in danger of adding wrong records to the dependency set. For example, ENOENT as the return value of an access system call indicates that the requested path does not exist or is a dangling symbolic link. Therefore, we cannot simply generate dependency records for the entire path. Instead, we determine which part of the path exists and add appropriate dependency records for only that part.

Usually, the main process creates child processes using fork. Our parser tracks dependency sets for the descendants of a traced process in order to generate a good signature. For example, a make process forks children to compile and link objects; if these child processes were omitted, the resulting dependency set would contain little useful information.

Initially, the parser sets the dependency set of a child process equal to the dependency set of its parent. It adds new records to the child's dependency set as the child executes. If the child communicates to its parent (e.g., by sending the parent a signal when it exits), the parser sets the dependency set of the parent process to be the union of the parent's current dependency set and the child's dependency set. The fork system call is usually followed by an exec system call that replaces the memory image of the process with one from an executable file. When this happens, the parser adds the executable file to the process's dependency set.

In our current implementation, the parser uses full path information for files and directories. We also considered using only the name of a file or directory instead of the whole path. However, our experiments revealed that the former method was slightly superior, mainly due to false matches between files with the same name but different paths. We did find that using only the file name was especially useful for shared libraries, because the location of libraries can vary widely across platforms. Therefore, our implementation uses only the file name for shared libraries. Our parser has one further optimizations: if an object being read is referred to by a symlink, the parser follows the symlink to also add entries for the real path of the object.

To diagnose a configuration error on a sick computer, our tool runs each predicate, traces its output, and generates its dependency set. It compares the dependency sets with those generated on the reference computer for each known bug. To compare dependency sets, the tool calculates the edit distance between the sets for each predicate. For each known bug, it sums the edit distances to calculate the similarity between the state of the sick computer and the state of the reference computer. It identifies the bug with the lowest total as the most likely diagnosis; in the case of ties, it reports all tied bugs as being equally likely to be the root cause.

5 Evaluation

Our evaluation measures how effectively our proposed dependency set method diagnoses configuration bugs using both targeted and "kitchen sink" predicates.

5.1 Methodology

In previous work [8], we developed a benchmark consisting of three applications: the CVS version control system, the gcc cross compiler and the Apache Web server. For each application, the benchmark consists of 10 common configuration bugs. It also contains 5–6 targeted predicates for each application such that each bug causes at least one predicate to fail. Although a complete description of our benchmark is omitted due to space constraints, Table 1 shows some examples of bugs and predicates. In addition, for each application we created a single "kitchen sink" predicate that detects all bugs.

In order to measure how sensitive our dependency set method is to variation across operating systems and installed software, we ran our experiments on four computers running different operating systems: Red Hat Enterprise Linux 3, Fedora core release 6, Ubuntu version 7.04, and FreeBSD version 6.2. Although these platforms are fairly similar in overall behavior, the execution signatures revealed a lot of subtle differences. For instance, in our Ubuntu platform libraries are located in "/lib/tls/i686", while in other systems "/lib" contains the libraries. As another example, FreeBSD

Application	Configuration problem descriptions
CVS	Setgid bit not set on repository, so group for new files is incorrect
	$CVSROOT misconfigured for a CVS user
GCC	Cross-compiler not configured for -pthread flag
	Cross-compiler not configured to pass the static link flag to the linker
Apache	Apache configuration does not allow CGI execution in user's home directory
	Apache not configured to load PHP module
Application	Predicate descriptions
CVS	a user checks in a project and checks it out again
	a user checks in a project, and a different user checks it out
GCC	compile a .c file and statically link in a math library
	take a multi-threaded .c file, compile it for the XScale architecture
Apache	`wget` a CGI script from a user's home directory
	`wget` the result of a PHP test page

Table 1. Example bugs and predicates

uses "/etc/pwd.db" and "/etc/spwd.db" for authentication, while other platforms use "/etc/passwd". We installed the same version of CVS and the gcc cross compiler on all machines. For Apache, we used version 2.0.50 for all machines, except for FreeBSD, which runs 2.0.59. The version of the PHP module that we used is 4.4.6, except for Fedora, which runs 4.4.7.

We used the Red Hat machine as the reference computer. For each application, we injected each bug. We then executed the targeted predicates and recorded the success or failure of each one, as well as its dependency set. We also executed the "kitchen sink" predicate for each bug, recording its outcome and dependency set.

We emulated sick computers by injecting each bug into all four computers. For each bug, we ran the targeted and "kitchen sink" predicates on each sick computer and used both the Hamming distance and dependency set methods to diagnose the bug. Each method returns a set of bugs that are judged to be the root cause of the configuration problem. Multiple bugs are returned by each method only in the case of ties, where each bug is judged equally likely to be the root cause. Two bugs of the benchmark (CVS bug 4 and Apache bug 4) were not applicable to FreeBSD platform due to differences in platform default behavior and application versions, so we omitted these bugs from our results.

We evaluated our results using two metrics from the information retrieval literature: precision and recall. Precision, which is the percent of false positives, is calculated as $|R \cap C|/|R|$, where R is the set of bugs returned by a method and C is the set of bugs that are the correct root cause. Recall, which is the percent of false negatives, is calculated as $|R \cap C|/|C|$.

5.2 Results

Table 2 shows results for the targeted predicates. We only show precision in the table since both the Hamming distance and dependency set methods have a recall of 100%, i.e., there were no false negatives in our experiments. Because the Hamming distance method only considers the success or failure of predicates, its results are the same on all sick computers. Therefore, we only show its precision once in the third column of the table. The remaining columns show the precision of the dependency set method on each sick computer.

As the third column of Table 2 shows, the Hamming distance method performs fairly well as long as an expert has taken the time to write targeted test cases. However, this method only considers the success or failure of predicate execution. Therefore, it cannot distinguish between situations with identical fail/pass patterns. Although our benchmark consists of targeted predicates, the Hamming distance algorithm still generates many ties. Across all bugs, its average precision is 57%.

As the remaining columns in the table show, the dependency set method has greater precision. On the Red Hat platform, the sick computer is identical to the reference computer. Thus, the dependency set method acts like an oracle, having precision of 100% for all bugs. For the remaining platforms, the dependency set method has average precision of 93%.

Table 3 shows results for the "kitchen sink" predicates. As before, neither method generates false negatives. However, the Hamming distance method has low precision for all bugs. It does not provide any useful information because kitchen sink predicates always fail. In contrast, the dependency set method is able to diagnose bugs much more accurately. The average precision of the dependency set method ranges from 93% to 100%, compared to 10% for the Hamming distance method. These results show that the dependency set method can still do an excellent job of diagnosing bugs without requiring the time-consuming task of writing targeted predicates.

The overhead of generating dependency sets is very small. On average, it takes less than 0.2 seconds to generate a signature from each trace output. Overall, it takes less than 14 seconds for CVS, 11 seconds for gcc and 27 seconds for Apache to run all the predicates under `strace` and generate a complete signature. In our experiments, the time required to compare the complete signature of a sick computer against the reference computer is less than 0.5 seconds. As the number of predicates and bugs in the database increases, the time required for generating the complete signature and comparing against the reference machine increases as well.

The accuracy of our method is dependent on the distance between bugs rather than the size of bug database. In other words, our method cannot accurately distinguish between bugs that are subtly different from each other and cause predicates to have similar executions. Although the chance of having such bugs increases as the database grows, the size of the database does not solely determine the precision of our method.

6 Related work

To the best of our knowledge, this work is the first to use the causal dependencies of predicate execution to

Application	Bug	Hamming distance	Dependency set (RHEL 3)	Dependency set (Fedora)	Dependency set (Ubuntu)	Dependency set (FreeBSD)
CVS	1	100%	100%	100%	100%	100%
	2	33%	100%	50%	50%	50%
	3	100%	100%	100%	100%	100%
	4	33%	100%	100%	100%	N/A
	5	100%	100%	100%	100%	100%
	6	33%	100%	100%	100%	100%
	7	33%	100%	50%	50%	50%
	8	33%	100%	50%	50%	50%
	9	100%	100%	100%	100%	100%
	10	33%	100%	50%	50%	50%
gcc	1	50%	100%	100%	100%	100%
	2	50%	100%	100%	100%	100%
	3	100%	100%	100%	100%	100%
	4	33%	100%	100%	100%	100%
	5	33%	100%	100%	100%	100%
	6	100%	100%	100%	100%	100%
	7	50%	100%	100%	100%	100%
	8	50%	100%	100%	100%	100%
	9	100%	100%	100%	100%	100%
	10	33%	100%	100%	100%	100%
Apache	1	100%	100%	100%	100%	100%
	2	100%	100%	100%	100%	100%
	3	20%	100%	100%	100%	100%
	4	20%	100%	100%	100%	N/A
	5	20%	100%	100%	100%	100%
	6	50%	100%	100%	100%	100%
	7	50%	100%	100%	100%	100%
	8	100%	100%	100%	100%	100%
	9	20%	100%	100%	100%	100%
	10	20%	100%	100%	100%	100%

Table 2. Precision of bug diagnoses for targeted predicates

Application	Bug	Hamming distance	Dependency set (RHEL 3)	Dependency set (Fedora)	Dependency set (Ubuntu)	Dependency set (FreeBSD)
CVS	1	10%	100%	100%	100%	100%
	2	10%	100%	100%	100%	50%
	3	10%	100%	100%	100%	100%
	4	10%	100%	100%	100%	N/A
	5	10%	100%	100%	100%	100%
	6	10%	100%	100%	100%	100%
	7	10%	100%	50%	50%	50%
	8	10%	100%	50%	50%	50%
	9	10%	100%	100%	100%	100%
	10	10%	100%	50%	50%	50%
gcc	1	10%	100%	100%	100%	100%
	2	10%	100%	100%	100%	100%
	3	10%	100%	100%	100%	100%
	4	10%	100%	100%	100%	100%
	5	10%	100%	100%	100%	100%
	6	10%	100%	100%	100%	100%
	7	10%	100%	100%	100%	100%
	8	10%	100%	100%	100%	100%
	9	10%	100%	100%	100%	100%
	10	10%	100%	100%	100%	100%
Apache	1	10%	100%	100%	100%	100%
	2	10%	100%	100%	100%	100%
	3	10%	100%	100%	100%	100%
	4	10%	100%	100%	100%	N/A
	5	10%	100%	100%	100%	100%
	6	10%	100%	100%	100%	100%
	7	10%	100%	100%	100%	100%
	8	10%	100%	100%	100%	100%
	9	10%	100%	100%	100%	100%
	10	10%	100%	100%	100%	100%

Table 3. Precision of bug diagnoses for kitchen sink predicates

diagnose configuration bugs. Previous systems have used predicates to help fix buggy computers. Chronus [11] also uses user-defined predicates to test the behavior of the system. Chronus tries to find the point in time where a system ceased to operate correctly by testing a predicate against different virtual machine snapshots. The success or failure of the predicate is assumed to precisely diagnose the bug. We must avoid this assumption in order to eliminate having an expert write a targeted predicate for each new bug. Since Chronus compares the system against itself, it is able to diagnose unknown bugs. Our method, however, cannot diagnose bugs that do not exist in the reference computer database. Our previous work, AutoBash [8], also used predicates but employed the Hamming distance method discussed in the paper.

PeerPressure [9] and its predecessor, Strider [10], also address the configuration management problem. These tools apply statistical methods to diagnose and fix configuration problems. PeerPressure and Strider benefit from the known schema of the registry, but cannot detect configuration errors that lie outside the registry. Our approach of analyzing predicate causality is more general and holds promise for dealing with errors that lie outside the registry and on other operating systems such as Unix variants. We assume that the bug is already known and exists in the reference computer database, but PeerPressure and Strider do not have this assumption.

Clarify [3] uses a similar approach of generating signatures that are based on program behavior. Clarify targets improved error reporting rather than configuration management. Clarify generates signatures using program features such as function call counts, call sites, and stack dumps. It then classifies the signatures using machine learning techniques. In contrast, our approach uses causal dependency information from a tree of processes to generate a signature.

Similar to our method, Yuan *et al.* [12] leverage system call information to diagnose configuration bugs. They correlate system call traces to problem root causes using machine learning techniques. To reduce system call variations, they use cross-time and cross-machine noise filtering techniques. Our method generates more robust signatures by extracting dependency sets from system call traces. The dependency set method does not need cross-time filtration and is accurate across variations of Unix operating systems.

Many prior systems use causality analysis. For instance, BackTracker [6] traces causality to determine what state has been changed during an intrusion. Aguilera *et al.* [1] trace RPCs to debug performance problems. PASS [7] uses causality to annotate files with provenance that describes their causal inputs. Our dependency sets capture similar information, but limit the scope of information collected to specific periods of time.

7 Conclusion

Configuration management is a difficult problem that is taking on increased importance as the complexity of modern computer systems grows. This paper contributes a novel method for error diagnosis that uses the causal dependencies of test case execution to detect similarities between a configuration state on a sick computer and another on a reference computer. We show that such information can be collected using only pre-existing system call tracing tools and without requiring application or operating system modification. As future work, we would like to measure the robustness of our signatures across different versions of the same application.

Acknowledgments

We thank Ya-Yunn Su and Kaushik Veeraraghavan for comments on this paper. This work has been supported by the National Science Foundation under award CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 74–89.

[2] BROWN, A. B., AND PATTERSON, D. A. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Technical Conference* (San Antonio, TX, June 2003).

[3] HA, J., ROSSBACH, C. J., DAVIS, J. V., ROY, I., RAMADAN, H. E., PORTER, D. E., CHEN, D. L., AND WITCHEL, E. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007* (San Diego, CA, 2007).

[4] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of Computer Security 6*, 3 (1998), 151–180.

[5] HOLLAND, D. A., JOSEPHSON, W., MAGOUTIS, K., SELTZER, M., STEIN, C., AND LIM, A. Research issues in no-futz computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 106–110.

[6] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.

[7] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[8] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, October 2007), pp. 237–250.

[9] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 245–257.

[10] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A blackbox, state-based approach to change and configuration management and support. In *Proceedings of Usenix Large Installation Systems Administration Conference* (October 2003), pp. 159–172.

[11] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 77–90.

[12] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated known problem diagnosis with event traces. In *Proceedings of EuroSys 2006* (Leuven, Belgium, 2006).