SYSADMIN

Automating Configuration Troubleshooting with ConfAid

MONA ATTARIYAN AND JASON FLINN

Mona Attariyan is a PhD candidate at the Computer Science and Engineering Department of the University of Michigan. Her research interests broadly include software systems, especially operating systems and distributed systems.
monattar@umich.edu

Jason Flinn is an associate professor of computer science and engineering at the University of Michigan. His research interests include operating systems, distributed systems, storage, and mobile computing.
jflinn@umich.edu

Complex software systems are difficult to configure and manage. When problems inevitably arise, operators spend considerable time troubleshooting those problems by identifying root causes and correcting them. The cost of troubleshooting is substantial. Technical support contributes 17% of the total cost of ownership of today's desktop computers [3], and troubleshooting misconfigurations is a large part of technical support. Even for casual computer users, troubleshooting is often enormously frustrating.

Our research group is exploring how operating system support for dynamic information flow analysis can substantially simplify and reduce the human effort needed to troubleshoot software systems. We are focusing specifically on configuration errors, in which the application code is correct, but the software has been installed, configured, or updated incorrectly so that it does not behave as desired. For instance, a mistake in a configuration file may lead software to crash, assert, or simply produce erroneous output.

Consider how users and administrators typically debug configuration problems. Misconfigurations are often exhibited by an application unexpectedly terminating or producing undesired output. While an ideal application would always output a helpful error message when such events occur, it is unfortunately the case that such messages are often cryptic, misleading, or even non-existent. Thus, the person using the application must ask colleagues and search manuals, FAQs, and online forums to find potential solutions to the problem.

ConfAid helps mitigate such problems. ConfAid is run offline, once erroneous behavior has been observed. A ConfAid user reproduces the problem by executing the application while ConfAid monitors the application's behavior. The user specifies the application she wishes to troubleshoot and its sources of configuration data (e.g., httpd.conf for the Apache Web server). ConfAid automatically diagnoses the root causes of self-evident errors, such as assertion failures and exits with non-zero return codes. ConfAid also allows its user to specify undesired output (e.g., specific error strings); it monitors application output to files, network, and other external devices for such user-specified error conditions.

When ConfAid observes erroneous application behavior, it outputs an ordered list of probable root causes. Each entry in the list is a token from a configuration source; our results show that ConfAid typically outputs the actual root cause as the first or second entry in the list. This allows the ConfAid user to focus on one or two specific configuration tokens when deciding how to fix the problem, which can dramatically improve the total time to recovery for the system.

The rest of this article briefly describes ConfAid's design and implementation, and it gives a short summary of our experiments with ConfAid. More details can be found in our OSDI '10 paper [1].

Design Principles of ConfAid

We begin by describing ConfAid's design principles.

Use White-Box Analysis

The genesis of ConfAid arose from AutoBash [8], our prior work in configuration troubleshooting. AutoBash tracks causality at process and file granularity in order to diagnose configuration errors. It treats each process as a black box, such that all outputs of the process are considered to be dependent on all prior inputs. We found AutoBash to be very successful in identifying the root cause of problems, but the success was limited in that AutoBash would often identify a complex configuration file, such as Apache's httpd.conf, as the source of an error. When such files contain hundreds of options, the root cause identification of the entire file is often too nebulous to be of great use.

Our take-away lessons from AutoBash were: (1) causality tracking is an effective tool for identifying root causes, and (2) causality should be tracked at a finer granularity than an entire process to troubleshoot applications with complex configuration files. These observations led us to use a white box approach in ConfAid that tracks causality within each process at byte granularity.

Operate on Application Binaries

We next considered whether ConfAid should require application source code for operation. While using source code would make analysis easier, source code is unavailable for many important applications, which would limit the applicability of our tool. Also, we felt it likely that we would have to choose a subset of programming languages to support, which would also limit the number of applications we could analyze. For these reasons, we decided to design ConfAid to not require source code; ConfAid instead operates on program binaries.

Embrace Imprecise Analysis

Our final design decision was to embrace an imprecise analysis of causality that relies on heuristics rather than using a sound or complete analysis of information flow. Using an early prototype of ConfAid, we found that for any reasonably complex configuration problem, a strict definition of causal dependencies led to our tool outputting almost all configuration values as the root cause of the problem. Thus, our current version of ConfAid uses several heuristics to limit the spread of causal dependencies. For instance, ConfAid does not consider all dependencies to be equal. It considers data flow dependencies to be more likely to lead to the root cause than control flow dependencies. It also considers control flow dependencies introduced closer to the error exhibition to be more likely to lead to the root cause than more distant ones. In some cases, ConfAid's heuristics can lead to false negatives and false positives. However, our results show that in most cases, they are quite effective in narrowing the search for the root cause and reducing execution time.

ConfAid's Information Flow Analysis

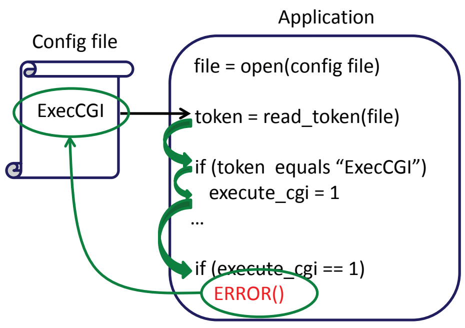

Figure 1: ConfAid propagates configuration tokens throughout the application using information flow analysis. When an error happens, ConfAid uses the propagated information to determine the root cause of the undesired outcome.

We use the example in Figure 1 to illustrate the mechanics of ConfAid. Assume that the application exhibits an error if the configuration token ExecCGI exists in the config file. When the application runs, ConfAid uses taint tracking [7] to dynamically monitor the propagation of configuration tokens and to determine how the erroneous outcome depends on the configuration data. When the application reads the value of the ExecCGI token from the configuration file, ConfAid taints the memory location that stores that value of token to indicate that its value could change if the user were to modify the value of ExecCGI in the configuration file. As the application executes, ConfAid observes that the value of execute_cgi depends on the value of the token, so it also taints that memory location. When the error happens, ConfAid sees that the error could have been avoided if the branch that tests execute_cgi had a different outcome. Since execute_cgi is tainted by the ExecCGI option, ConfAid identifies that configuration option as the root cause of the error.

To analyze the information flow, ConfAid adds custom logic, referred to as instrumentation, to each application binary using Pin [6]. The instrumentation monitors each system call, such as read or pread, that could potentially read data from a configuration source. If the source of the data returned by a system call is a configuration file, ConfAid annotates the registers and memory addresses modified by the system call with a marker that indicates a dependency on a specific configuration token. Borrowing terminology from the taint tracking literature, we refer to this marking as the taint of the memory location. If an address or register is tainted by a token, ConfAid believes that the value at that location might be different if the value of the token in the original configuration source were to change.

```
/* a, b, c and d are read from the config file*/
if (c == 0) { /* c set to 0 in config file */
       x = a;  /* taken path */
} else {
       y = b;  /* alternate path */
}
z = d;
if (z) assert();  /* The erroneous behavior */
```

Figure 2: Example to illustrate causality tracking. The assertion only depends on variable z, which itself depends on the value of configuration token d. Configuration token c only affects variables x and y.

ConfAid specifies the taint of each variable as a set of configuration options. For instance, if the taint set of a variable is { FOO, BAR }, ConfAid believes that the value of that variable could change if the user were to modify either the FOO or the BAR token in the configuration file.

Taint is propagated via data flow and control flow dependencies. When a monitored process executes an instruction that modifies a memory address, register, or CPU flag, the taint set of each modified location is set to the union of the taint sets of the values read by the instruction. For example, consider the instruction x = y + z where the taint set of x becomes the union of taint sets of y and z. Intuitively, the value of x might change if a configuration token were to cause y or z to change prior to the execution of this instruction.

In traditional taint tracking for security purposes, control flow dependencies are often ignored to improve performance because they are harder than data flow dependencies for an attacker to exploit. With ConfAid, however, we have found that tracking control flow dependencies is essential since they propagate the majority of configuration-derived taint. A naive approach to tracking control flow is to union the taint set of a branch conditional with a running control flow dependency for the program. However, without mechanisms to remove control flow taint, the taint grows without limit. This causes too many false positives in ConfAid's root cause list.

A more precise approach takes into account the basic block structure of a program. Consider the example in Figure 2. Assume a, b, c, and d were read from a configuration file and have taint sets assigned to them. The value of c does not affect whether the last two statements are executed, since they execute in all possible paths (and therefore for all values of c). Thus, the taint of c should be removed from the control flow taint before executing z = d. When the program asserts, the control flow taint should only include the taint of d to correctly indicate that changing the value of d might fix the problem.

ConfAid also tracks implicit control flow dependencies. In Figure 2, the values of x and y depend on c when the program asserts, since the occurrence of their assignments to a and b depend on whether or not the branch is taken. Note that y is still dependent on c even though the else path is not taken by the execution, since the value of y might change if a configuration token is modified such that the condition evaluates differently.

When the program executes a branch with a tainted condition, ConfAid first determines the merge point (the point where the branch paths converge) by consulting the control flow graph. Prior to dynamic analysis, ConfAid obtains the graph by using IDA Pro [2] to statically analyze the executable and any libraries it uses (e.g., libc and libssl). For each tainted branch, ConfAid next explores each alternate path that leads to the merge point. We define an alternate path to be any path not taken by the actual program execution that starts at a conditional branch instruction for which the branch condition is tainted by one or more configuration values. ConfAid uses alternate path exploration to learn which variables would have been assigned had the condition evaluated differently due to a modified configuration value.

To evaluate an alternate path, ConfAid switches the condition outcome and forces the program to execute the alternate path. ConfAid uses copy-on-write logging to checkpoint and roll back application state. When a memory address is first altered along an alternate path, ConfAid saves the previous value in an undo log. At the end of the execution, application state is replaced with the previous values from the log. Many branches need not be explored since their conditions are not tainted by any configuration token. After exploring the alternate paths, ConfAid performs a similar analysis for the path actually taken by the program. This is the actual execution, so no undo log is needed.

ConfAid also uses alternate path exploration to learn which paths avoid erroneous application behavior. An alternate path is considered to avoid the erroneous behavior if the path leads to a successful termination of the program or if the merge point of the branch occurs after the occurrence of the erroneous behavior in the program (as determined by the static control flow graph). ConfAid unions the taint sets of all conditions that led to such alternate paths to derive its final result. This result is the set of all configuration tokens which, if altered, could cause the program to avoid the erroneous behavior.

Figure 3 shows four examples that illustrate how ConfAid detects alternate paths that avoid the erroneous behavior. In case (a), the error occurs after the merge point of the conditional branch. ConfAid determines that the branch does not contribute to the error, because both paths lead to the same erroneous behavior. In case (b), the alternate path avoids the erroneous behavior because the merge point occurs after the error, and the alternate path itself does not exhibit any other error. In this case, ConfAid considers tokens in the taint set of the branch condition as possible root causes of the error, since if the application had taken the alternate path, it could have avoided the error. In case (c), the alternate path leads to a different error (an assertion). Therefore, ConfAid does not consider the taint of the branch as a possible root cause, because the alternate path would not lead to a successful termination. In case (d), there are two alternate paths, one of which leads to an assertion and one that reaches the merge point. In this case, since there exists an alternate path that avoids the erroneous behavior, configuration tokens in the taint set of the branch condition are possible root causes.

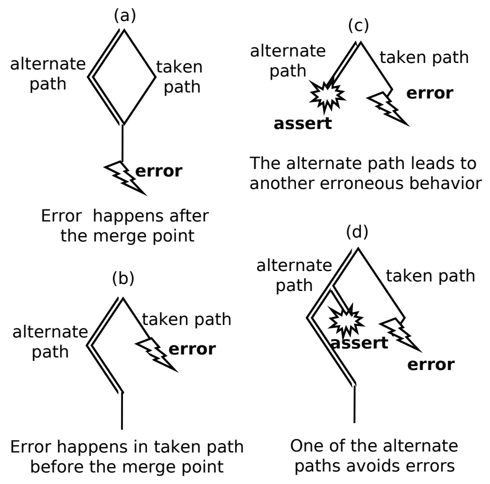
(a)
alternate path
taken path
error
Error happens after the merge point
(c)
alternate path
taken path
assert
error
The alternate path leads to another erroneous behavior
(b)
alternate path
taken path
error
Error happens in taken path before the merge point
(d)
alternate path
taken path
assert
error
One of the alternate paths avoids errors

Figure 3: Examples illustrating ConfAid path analysis

Heuristics for Performance

ConfAid uses two heuristics to simplify control flow analysis. The first heuristic is the bounded horizon heuristic. ConfAid only executes each alternate path for a fixed number of instructions. By default, ConfAid uses a limit of 80 instructions. All addresses and registers modified within the limit are used to calculate information flow dependencies after the merge point. Locations modified after the limit do not affect dependencies introduced at the merge point.

The second heuristic simplifies control flow analysis by assuming that the configuration file contains only a single error—we refer to this as the single mistake heuristic. This heuristic reduces the amount of taint in the application and the number of alternative paths that need to be explored by restricting the number of configuration values that can change. The single mistake heuristic may lead to false negatives. Potentially, if ConfAid cannot find a root cause, we can relax the single-mistake assumption by allowing ConfAid to assume that two or more tokens are erroneous. In our experiments to date, this heuristic has yet to trigger a false negative.

Heuristics for Reducing False Positives

In our design as described so far, two configuration tokens are considered equal taint sources even if one has a direct causal relationship to a location (e.g., the value in memory was read directly from the configuration file) and another has a nebulous relationship (e.g., the taint was propagated along a long chain of conditional assignments deep along alternate paths). Another problem we noticed was that loops could cause a location to become a global source and sink for taint. For instance, Apache reads its configuration values into a linked list structure and then traverses the list in a loop to find the value of a particular configuration token. During the traversal, the program control flow picks up taint from many configuration options, and these taints are sometimes transferred to the configuration variable that is the target of the search.

We realized that both of these problems were caused by our implicit assumption that all information flow relationships should be treated equally. Based on this observation, we decided to modify our design to instead track taint as a floating-point weight ranging in value between zero and one. When the error happens, ConfAid can rank the possible root causes based on their weights with the option with the highest weight being ranked first.

Our weights are based on two heuristics. First, data flow dependencies are assumed to be more likely to lead to the root cause than control flow dependencies. Second, control flow dependencies are assumed to be more likely to lead to the root cause if they occur later in the execution (i.e., closer to the erroneous behavior). Specifically, we assign taints introduced by control flow dependencies only half the weight of taints introduced by data flow dependencies. Further, each nested conditional branch reduces the weight of dependencies introduced by prior branches in the nest by one half. We chose a weight of 0.5 for speed: it can be implemented efficiently with a vector bit shift.

Multi-Process Causality Tracking

The most difficult configuration errors to troubleshoot involve multiple interacting processes. Such processes may be on a single computer, or on multiple computers connected by a network. To troubleshoot such cases, ConfAid instruments multiple processes at the same time and propagates taint information at per-byte granularity along with the data sent when the processes communicate. ConfAid supports processes that communicate using UNIX sockets, pipes, and TCP and UDP sockets and files. Since these operations are performed by Pin instrumentation, the taint propagation is hidden from the application and no operating system modifications are needed.

Evaluation

Our evaluation answers two questions:

> How effective is ConfAid in identifying the root cause of configuration problems?
>
> How long does ConfAid take to find the root cause?

Experimental Setup

We evaluated ConfAid on three applications: the OpenSSH server version 5.1, the Apache HTTP server version 2.2.14, and the Postfix mail transfer agent version 2.7. All of our experiments were run on a Dell OptiPlex 980 desktop computer with an Intel Core i5 Dual Core processor and 4GB of memory. The machine runs Linux kernel version 2.6.21. For Apache, ConfAid instruments a single process; for OpenSSH, up to two processes; and for Postfix, up to six processes.

To evaluate ConfAid, we manually injected errors into correct configuration files. Then we ran a test case that caused the error we injected to be exhibited. We used ConfAid to instrument the process (or processes) for that application and obtained the ordered list of root causes found by ConfAid. We use two metrics to evaluate ConfAid's effectiveness: the ranking of the actual root cause, i.e., the injected mistake, in the list returned by ConfAid and the time to execute the instrumented application.

We used two different methods to generate configuration errors. First, we injected 18 real-world configuration errors that were reported in online forums, FAQ pages, and application documentation. Second, we used the ConfErr tool [4] to inject random errors into the configuration files of the three applications. ConfErr uses human error models rooted in psychology and linguistics to generate realistic configuration mistakes. We used ConfErr to produce 20 errors for each application.

Results

Application	*Root causes ranked first*	*Root causes ranked first with one tie*	*Root causes ranked second*	*Root causes ranked second with one tie*	*Avg. time to run*
OpenSSH (7 bugs)	2	2	2	1	52s
Apache (6 bugs)	3	1	0	2	2m 48s
Postfix (5 bugs)	5	0	0	0	57s

Table 1: Results for real-world configuration bugs

Table 1 summarizes our results for real-world misconfigurations. ConfAid ranks the actual root cause first in 13 cases and second in the other 5. Sometimes, when the actual root cause is ranked second, the token ranked first provides a valuable clue to help troubleshoot the problem. For instance, in Apache the actual error usually occurs nested inside a section or directive command in the config file. For the two Apache errors where the root cause is ranked second, the top-ranked option is the section or directive containing the error.

ConfAid's average execution time of 1:32 minutes is much faster and far less frustrating than manual troubleshooting. For instance, one of the Apache misconfigurations is taken from a thread in linuxforums.org [5]. After trying to fix the misconfiguration for quite a while, the user went to the trouble of posting the question in the forum and waited two days for an answer. ConfAid identified the root cause in less than three minutes.

Application	*Root causes ranked first*	*Root causes ranked first with one tie*	*Root causes ranked second*	*Root causes ranked second with one tie*	*Avg. time to run*	*Avg. time to run*
OpenSSH	17 (85%)	1 (5%)	1 (5%)	0	1 (5%)	7s
Apache	17 (85%)	1 (5%)	0	1 (5%)	1 (5%)	24s
Postfix	15 (75%)	0	2 (10%)	0	3 (15%)	38s

Table 2: Results for randomly generated bugs

Table 2 summarizes the results for randomly generated configuration errors. For OpenSSH, ConfAid ranked the root cause first or second for 95% of the bugs. For the last bug, ConfAid could not run to completion due to unsupported system calls used in the code path. We could remedy this by supporting more calls. ConfAid also successfully diagnosed 95% of the Apache errors. For the remaining bug, the correct root cause was ranked 9th due to our weighting heuristic. For Postfix, ConfAid diagnosed 85% of the errors effectively. The remaining three errors were due to missing configuration options. Currently, ConfAid only considers all tokens present in the configuration file as possible sources of the root cause. If a default value can be overridden by a token not actually in the file, then ConfAid will not detect the missing token as a possible root cause. We plan to extend ConfAid to also diagnose misconfigurations that are due to missing configuration tokens.

Conclusions and Future Work

Configuration errors are costly, time-consuming, and frustrating to troubleshoot. ConfAid makes troubleshooting easier by pinpointing the specific token in a configuration file that led to an erroneous behavior. Compared to prior approaches, ConfAid distinguishes itself by analyzing causality within processes as they execute without the need for application source code. It propagates causal dependencies among multiple processes and outputs a ranked list of probable root causes. Our results show that ConfAid usually lists the actual root cause as the first or second entry in this list. Thus, ConfAid can substantially reduce total time to recovery and perhaps make configuration problems a little less frustrating.

There are several possible directions for future work. First, ConfAid currently only troubleshoots configuration problems that lead to crashes, assertion failures, and incorrect output; it does not yet help diagnose misconfigurations that cause poor performance. One approach to tackling performance problems that we are investigating is to first use statistical sampling to associate use of a bottleneck resource such as disk or CPU with specific points in the program execution. Then,ConfAid-style analysis can determine which configuration tokens most directly affect the frequency of execution of those points.

Second, ConfAid currently assumes that the configuration file contains only one erroneous token. If fixing a particular error requires changing two tokens, then ConfAid's alternate path analysis may not identify both tokens. We therefore plan to allow ConfAid to track sets of two or more misconfigured tokens and measure the resulting performance overhead. Potentially, we can use an expanding search technique in which ConfAid initially performs an analysis assuming only a single mistake, and then performs a lengthier analysis allowing multiple mistakes if the first analysis does not yield satisfactory results.

Finally, we believe that ConfAid can be best improved if it is used and tested by many people. Therefore, we plan to release an open source version of ConfAid to the public. This will require us to make ConfAid more robust in diverse computing environments, and we will also need to use an open source static analysis tool to generate a control flow graph.

Acknowledgments

This research was supported by NSF award CNS-1017148. 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] M. Attariyan and J. Flinn, "Automating Configuration Troubleshooting with Dynamic Information Flow Analysis," *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, October 2010.

[2] IDA Pro disassembler: http://www.hex-rays.com/idapro.

[3] A. Kapoor, "Web-to-Host: Reducing Total Cost of Ownership," Technical Report 200503, The Tolly Group, May 2000.

[4] L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A Tool for Assessing Resilience to Human Configuration Errors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2008, pp. 157–166.

[5] http://www.linuxforums.org/forum/servers/125833-solved-apache-wont-follow-symlinks.html.

[6] C.-K. 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005, pp. 190–200.

[7] J. Newsome and D. Song, "Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software," *Proceedings of the 12th Network and Distributed Systems Security Symposium*, February 2005.

[8] Y.-Y. Su, M. Attariyan, and J. Flinn, "AutoBash: Improving Configuration Management with Operating System Causality Analysis," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, October 2007, pp. 237–250.